Software Communications Architecture Specification

APPENDIX C CORE FRAMEWORK IDL

Revision Summary

1.0	Initial Release
1.1	Updated IDL to reflect SCAS changes made for v1.1; updated comments.
2.0	Incorporate approved Change Proposals, numbers 175, 245, 277, 278, 282, 311, 336, 345.
2.1	Incorporate approved Change Proposals, numbers 142, 175, 245, 277, 278, 282, 306, 311, 336, 345, 360.

Change Proposals are controlled by the JTRS Change Control Board. CPs incorporated into the SCA are considered "closed" and can be seen on the JTRS web site at: www.jtrs.sarda.army.mil/docs/documents/sca_ccb.html.

Table of Contents

APPENDIX C CORE FRAMEWORK IDL

The CF interfaces are expressed in CORBA IDL. The IDL has been generated directly by the Rational Rose UML software modeling tool. This "forward engineering" approach ensures that the IDL accurately reflects the architecture definition as contained in the UML models. Any IDL compiler for the target language of choice may compile the generated IDL.

The CF interfaces are contained in the CF CORBA module. Additionally, IDL modules are provided for interfaces that extend the *CF::Port* interface by defining basic data sequence types and for pushing data to a consumer or pulling data from a producer. The LogService CORBA Module contains the interfaces and types for a log service. Figure C-1 shows the relationship between these CORBA modules.

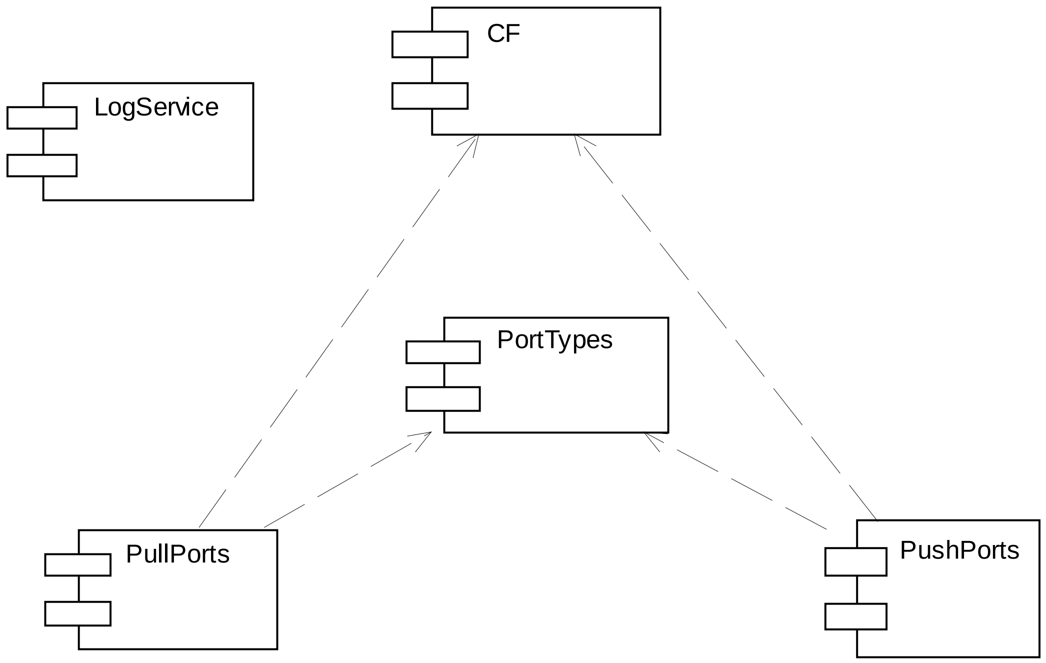

Figure C-1. Relationships Between CORBA Modules

The IDL modules are also available in electronic form.

C.1 CORE FRAMEWORK IDL.

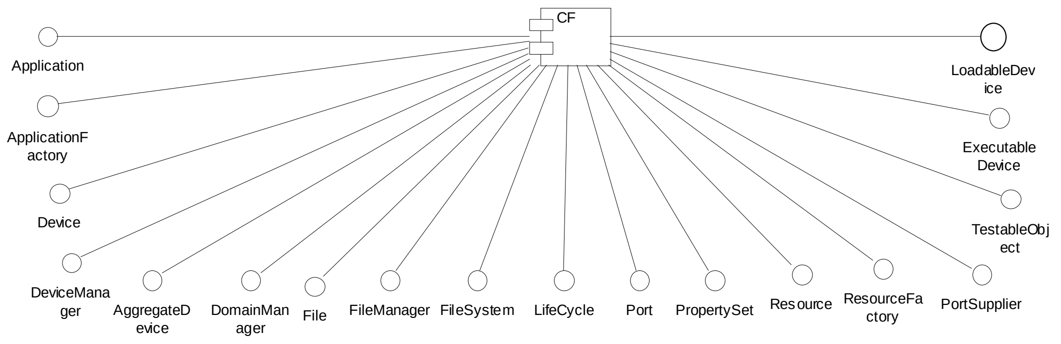

Figure C-2. CF CORBA Module

The following is the CF IDL generated from the Rational Rose model, version 2000e.

```
#ifndef __CF_DEFINED
#define __CF_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

/* This package provides the main framework for all objects within the radio.
*/

module CF {
      interface File;
      interface Resource;
      interface Application;
      interface Device;
      interface ApplicationFactory;
      interface DeviceManager;


      /* This type is a CORBA  IDL struct type which can be used to hold any
       CORBA basic type or static IDL type. */

      struct DataType {
            /* The id attribute indicates the kind of value and type
            (e.g., frequency, preset, etc.).  The id can be an UUID string,
             an integer string, or a name identifier. */
            string id;
            /* The value attribute can be any static IDL type or CORBA basic
            type. */
            any value;
      };

      /* This exception indicates an invalid file name was passed to a File
      Service operation.  The message provides information describing why the
      filename was invalid. */

      exception InvalidFileName {
```

```
      string msg;
};

/* This exception indicates a file-related error occurred. The message
shall provide information describing the error.  The message can be
used for logging the error. */

exception FileException {
      string msg;
      /* The error code that corresponds to the error message. */
      unsigned short errorCode;
};

/* This exception indicates an invalid component profile error. */

exception InvalidProfile {
};

/* The Properties is a CORBA IDL unbounded sequence of CF DataType(s),
which can be used in defining a sequence of name and value pairs. */

typedef sequence <DataType> Properties;

/* This exception indicates an invalid CORBA object reference error. */

exception InvalidObjectReference {
      string msg;
};

/* This type is a CORBA unbounded sequence of octets. */

typedef sequence <octet> OctetSequence;

/* This type defines a sequence of strings */

typedef sequence <string> StringSequence;

/* This exception indicates a set of properties unknown by the
component. */

exception UnknownProperties {
      Properties invalidProperties;
};

/* DeviceAssignmentType defines a structure that associates a component
with the Device upon which the component is executing on. */

struct DeviceAssignmentType {
      string componentID;
      string assignedDeviceID;
};

/* The IDL sequence, DeviceAssignmentSequence, provides a unbounded
sequence of 0..n of DeviceAssignmentType. */

typedef sequence <DeviceAssignmentType> DeviceAssignmentSequence;
```

```
/* This type defines an unbounded sequence of Devices.

The IDL to Ada mapping has a problem with self referential interfaces.
To get around this problem, the interface Device forward declaration
has been created and this type has been moved outside of the Device
interface. */

typedef sequence <Device> DeviceSequence;

/* This interface defines behavior for a Device that can be used for
adding and removing Devices from the Device. This new interface can be
provided via inheritance or as a "provides port"  for any Device that
is capable of an aggregate relationship. Aggregated Devices use this
interface to add or remove themselves from composite Devices when being
created or torn-down. */

interface AggregateDevice {
      /* The readonly devices attribute contains a list of devices that
      have been added to this Device or a sequence length of zero if
      the Device has no aggregation relationships with other Devices.
      */

      readonly attribute DeviceSequence devices;

      /* This operation provides the mechanism to associate a Device
      with another Device. Whena Device changes state or it is being
      torn down, this affects its associated Devices.

      The addDevice operation adds the input associatedDevice parameter
      to the AggregateDevice's devices attribute when the
      associatedDevice does not exist in the devices attribute.  The
      associatedDevice is ignored when duplicated.

      The addDevice operation writes a FAILURE_ALARM log record,upon
      unsuccessful adding of an associatedDevice to the
      AggregateDevice's devices attribute.

      This operation does not return any value.

      The addDevice operation raises the CF InvalidObjectReference when
      the input associatedDevice is a nil CORBA object reference.
      @roseuid 3A5DAE9102D6 */
      void addDevice (
            in Device associatedDevice
            )
            raises (InvalidObjectReference);

      /* This operation provides the mechanism to disassociate a Device
      with another Device.

      The removeDevice operation removes the input associatedDevice
      parameter from the AggregateDevice's devices attribute.

      The removeDevice operation writes a FAILURE_ALARM log record,
      upon unsuccessful removal of the associatedDevice from the
```

```
        AggregateDevice's devices attribute.



        This operation does not return any value.

        The removeDevice operation raises the CF InvalidObjectReference
        when the input associatedDevice is a nil CORBA object reference
        or does not exist in the aggregate Device's devices attribute.
        @roseuid 3A5DAE9102D8 */
        void removeDevice (
              in Device associatedDevice
              )
              raises (InvalidObjectReference);

  };

  /* The FileSystem interface defines the CORBA operations to enable
  remote access to a physical file system. */

  interface FileSystem {
        /* This exception indicates a set of properties unknown by the
        FileSystem object. */

        exception UnknownFileSystemProperties {
              Properties invalidProperties;
        };

        /* This constant indicates file system size. */

        const string SIZE = "SIZE";
        /* This constant indicates the available space on the file
        system. */

        const string AVAILABLE_SIZE = "AVAILABLE_SPACE";
        /* The remove operation removes the file with the given filename.
        This operation ensures that the filename is an absolute pathname
        of the file relative to the target FileSystem. If an error
        occurs, this operation raises the appropriate exception:

        CF InvalidFilename      - The filename is not valid.
        CF FileException        - A file-related error occurred during
                                the operation.
        @roseuid 364B4B2E26B0 */
        void remove (
              in string fileName
              )
              raises (FileException, InvalidFileName);

        /* The copy operation copies the source file with the specified
        sourceFileName to the destination file with the specified
        destinationFileName. This operation ensures that the
        sourceFileName and destinationFileName are absolute pathnames
        relative to the target FileSystem. If an error occurs, this
        operation raises the appropriate exception:
```

```
CF InvalidFilename      - The filename is not valid.
CF FileException        - A file-related error occurred during
                          the operation.
@roseuid 364B4B5A0640 */
void copy (
      in string sourceFileName,
      in string destinationFileName
      )
      raises (InvalidFileName, FileException);

/* The exists operation checks to see if a file exists based on
the filename parameter. This operation ensures that the filename
is a full pathname of the file relative to the target FileSystem
and raise an CF InvalidFileName exception if the name is invalid.

This operation shall return True if the file exists, otherwise
False shall be returned.
@roseuid 3665751C2AA0 */
boolean exists (
      in string fileName
      )
      raises (InvalidFileName);

/* The list operation returns a list of filenames based upon the
search pattern given.  The list operation supports the following
wildcard characters:
(1) * used to match any sequence of characters (including null).
(2) ? used to match any single character.

These wildcards may only be applied to the base filename in the
search pattern given.  For example, the following are valid
search patterns:
"/tmp/files/ *.*" Returns all files and directories within the
"/tmp/files" directory.  Directory names shall be indicated with
a "/" at the end of the name.

"/tmp/files/foo*" Returns all files beginning with the letters
"foo" in the "/tmp/files directory".

"/tmp/files/f??"  Returns all 3 letter files beginning with the
letter f in the "/tmp/files directory".

The list operation raises the CF InvalidFileName exception when
the input pattern does not start with a slash '/' or cannot be
interpreted due to unexpected characters."
@roseuid 36669644E5F0 */
StringSequence list (
      in string pattern
      )
      raises (InvalidFileName);

/* The create operation creates a new File based upon the
provided file name and returns a File to the opened file.  A null
file is returned and a related exception shall be raised if an
error occurs.
```

```
CF InvalidFilename      - The filename is not valid.
CF FileException        - File already exists or another file
                        error occurred.
@roseuid 36CAC30F37A8 */
File create (
      in string fileName
      )
      raises (InvalidFileName, FileException);

/* The open operation opens a file based upon the input fileName.
The read_Only parameter indicates if the file should be opened
for read access only.  When read_Only is false the file is opened
for write access.

The open operation returns a File component parameter on
successful completion.  The open operation returns a null file
component reference if the open operation is unsuccessful.  If
the file is opened with the read_Only flag set to true, then
writes to the file will be considered an error.

The open operation raises the CF FileException if the file does
not exist or another file error occurred.

The open operation raises the CF InvalidFilename exception when
the filename is not a valid file name or not an absolute
pathname.
@roseuid 36CAC3ECE2A0 */
File open (
      in string fileName,
      in boolean read_Only
      )
      raises (InvalidFileName, FileException);

/* The mkdir operation create a FileSystem directory based on the
directoryName given.  This operation creates all parent
directories required to create the directory path given. If an
error occurs, this operation raises the appropriate exception.

Exceptions/Errors
CF InvalidFilename      - The directory name is not valid.
CF FileException        - A file-related error occurred during
                        the operation.
@roseuid 388F55390C58 */
void mkdir (
      in string directoryName
      )
      raises (InvalidFileName, FileException);

/* The rmdir operation removes a FileSystem directory based on
the directoryName given. If an error occurs, this operation
raises the appropriate exception.

Exceptions/Errors
CF InvalidFilename      - The directory name is not valid.
CF FileException        - Directory does not exist or another
                        file-related error occurred.
```

```
        @roseuid 388F554033F8 */
        void rmdir (
              in string directoryName
              )
              raises (InvalidFileName, FileException);

        /* The query operation returns file system information to the
        calling client based upon the given fileSystemProperties' ID.

        As a minimum, the FileSystem query operation supports the
        following fileSystemProperties:
        1. SIZE - an ID value of "SIZE causes query to return an unsigned
        long long containing the file system size (in octets).
        2. AVAILABLE SPACE - an ID value of "AVAILABLE SPACE" causes the
        query operation to return an unsigned long long containing the
        available space on the file system (in octets).


        The query operation raises the UnknownFileSystemProperties
        exception when the given file system property is not recognized.
        @roseuid 389196D696B0 */
        void query (
              inout Properties fileSystemProperties
              )
              raises (UnknownFileSystemProperties);

  };

  /* The File interface provides the ability to read and write files
  residing within a CF compliant distributed FileSystem.  A file can be
  thought of conceptually as a sequence of octets with a current
  filepointer describing where the next read or write will occur.
  This filepointer points to the beginning of the file upon construction
  of the file object.  The File interface is modeled after the POSIX/C
  file interface. */

  interface File {
        /* This exception indicates an error occurred during a read or
        write operation to a File. The message provides information
        describing why the I/O error occurred. */

        exception IOException {
              /* The error code that corresponds to the error message. */
              unsigned short errorCode;
              string msg;
        };

        /* This exception indicates the file pointer is out of range
        based upon the current file size. */

        exception InvalidFilePointer {
        };

        /* The readonly fileName attribute contains the file name given
        to the FileSystem open/create operation.  The syntax for a
        filename is based upon the UNIX operating system.  That is, a
```

```
sequence of directory names separated by forward slashes (/)
followed by the base filename.  The fileName attribute will
contain the filename given to the FileSystem::open operation. */

readonly attribute string fileName;
/* The readonly filePointer attribute contains the file position
where the next read or write will occur. */

readonly attribute unsigned long filePointer;

/* The read operation reads octets from the file referenced up to
the number specified by the length parameter and change the value
of the filePointer attribute forward by the number of octets
actually read.  The read operation only reads less than the
maximum number of octets specified in the length parameter when
an end of file is encountered.

The read operation returns via the out Message parameter an CF
OctetSequence that equals the number of octets actually read from
the File.  If the filePointer attribute value reflects the end of
the File, the read operation returns a 0-length CF OctetSequence.

The read operation raises the IOException when a read error
occurs.
@roseuid 364B3D91DA40 */
void read (
      out OctetSequence data,
      in unsigned long length
      )
      raises (IOException);

/* The write operation writes data to the file referenced. If the
write is successful, the write operation shall increment the
filePointer attribute to reflect the number of octets written. If
the write is unsuccessful, the filePointer attribute value is
maintained or is restored to its value prior to the write
operation call.

This operation does not return any value.

The write operation raises the IOException when a write error
occurs.
@roseuid 364B3DA2AFD0 */
void write (
      in OctetSequence data
      )
      raises (IOException);

/* The sizeOf operation returns the current size of the file.
The CF FileException is raised when a file-releated error occurs
(e.g. the file does not exist anymore).
@roseuid 36AE182BBF90 */
unsigned long sizeOf ()
      raises (FileException);

/* The close operation releases any OE file resources associated
```

```
        with the component.  The close operation makes the file
        unavailable to the component

        A client should release its CORBA File reference after closing
        the File.  The close operation raises CF FileException exception
        when it cannot successfully close the file.
        @roseuid 388E0477F138 */
        void close ()
              raises (FileException);

        /* The setFilePointer operation positions the file pointer where
        the next read or write will occur.

        The setFilePointer operation sets the filePointer attribute value
        to the input filePointer.

        This operation does not return anyvalue.

        The setFilePointer operation raises the CF FileException when the
        File can not be successfully accessed to set thefilePointer
        attribute. The setFilePointer operation raises the
        InvalidFilePointer exception when the filePointer
        parameterexceeds the file size.
        @roseuid 39088B800D38 */
        void setFilePointer (
              in unsigned long filePointer
              )
              raises (InvalidFilePointer, FileException);

  };

  /* A ResourceFactory is used to create and tear down a Resource.  The
  ResourceFactory interface is designed after the Factory Design
  Patterns.  Each ResourceFactory object creates a specific type of
  Resource within the radio.  The ResourceFactory interface provides a
  one-step solution for creating a Resource, reducing the overhead of
  starting up Resources.  In CORBA, there are two separate object
  reference counts.  One for the client side and one for the server side.
  The Factory keeps a server-side reference count of the number of
  clients that have requested the resource.  When a client is done with a
  resource, the client releases the client resource reference and calls
  releaseResource to the ResourceFactory.  When the server-side reference
  goes to zero, the server resource object is released from the ORB that
  causes the resource to be destroyed. */

  interface ResourceFactory {
        /* This type defines the identity of a Resource created by the
        ResourceFactory. */

        typedef unsigned short ResourceNumType;

        /* This exception indicates the resource number does not exist in
        the ResourceFactory. */

        exception InvalidResourceNumber {
        };
```

```
/* This exception indicates that the shutdown method failed to
release the ResourceFactory from the CORBA environment due to the
fact the Factory still contains Resources.  The message is
component-dependent, providing additional information describing
why the shutdown failed. */

exception ShutdownFailure {
      /* This message indicates the reason for the shutdown
      failure. */
      string msg;
};

/* This operation provides the capability to create Resources in
the same process space as the ResourceFactory or to return a
Resource that has already been created.  This behavior is an
alternative approach for creating a Resource to the
Device::execute operations.

The ResourceNumber is the identifier for Resource.  The
qualifiers are parameter values used by the ResourceFactory in
creation of the Resource.  The ApplicationFactory can determine
the values to be supplied for the qualifiers from the description
in the ResourceFactory's Software Profile.  The qualifiers may be
used to identify, for example, specific subtypes of Resources
created by a ResourceFactory.

If no Resource existsfor the given resourceNumber, the
createResource operation creates a Resource.  If the Resource
already exists, that Resource is returned. The createResource
operation assigns the given resourceNumber to a new Resource and
set a reference count to one or, in the case that the Resource
already exists, increment the count by one.The reference count is
used to indicate the number of times that a specific Resource
reference has been given to requesting clients.  This ensures
that the ResourceFactory does not release a Resource that has a
reference count greater than 0.  (Multiple clients could request
the release of the Resource after obtaining a reference to the
Resource).

The createResource operation returns a reference to the created
Resource or the existing Resource. The createResource operation
returns a nil CORBA component reference when the operation is
unable to create or find the Resource.

This operation does not raise any exceptions.
@roseuid 356B1F02C620 */
Resource createResource (
      in ResourceNumType resourceNumber,
      in Properties qualifiers
      );

/* In CORBA there is client side and server side representation
of a Resource.  This operation provides the mechanism of
releasing the Resource in the CORBA environment on the server
side when all clients are through with a specific Resource.  The
```

```
        client still has to release its client side reference of the
        Resource.

        The releaseResource operation decrements the reference count for
        the specified resource, as indicated by the resourceNumber. The
        releaseResource operation makes the Resource no longer avaliable
        (ie, it is released from the CORBA environment) when the
        Resource's reference count is zero.

        This operation does not return a value.

        The releaseResource operation raises the InvalidResourceNumber
        exception if an invalid resourceNumber is received.
        @roseuid 356B1F4E9140 */
        void releaseResource (
              in ResourceNumType resourceNumber
              )
              raises (InvalidResourceNumber);

        /* In CORBA there is client side and server side representation
        of a ResourceFactory.  This operation provides the mechanism for
        releasing the ResourceFactory from the CORBA environment on the
        server side.  The client has the responsibility to release its
        client side reference of the ResourceFactory.

        The shutdown operation results in the ResourceFactory being
        unavailable to any subsequent calls to its object reference (i.e.
        it is released from the CORBA environment).

        This operation does not raise any exceptions.


        The shutdown operation raises the ShutdownFailure exception for
        any error that prevents the shutdown of the ResourceFactory.
        @roseuid 356C2593F700 */
        void shutdown ()
              raises (ShutdownFailure);

  };

  /* Multiple, distributed FileSystems may be accessed through a
  FileManager. The FileManager interface appears to be a single
  FileSystem although the actual file storage may span multiple physical
  file systems.  This is called a federated file system. A federated file
  system is created using the mount and unmount operations. Typically,
  the Domain Manager or system initialization software will invoke these
  operations.

  The FileManager inherits the IDL interface of a FileSystem. Based upon
  the pathname of a directory or file and the set of mounted filesystems,
  the FileManager will delegate the FileSystem operations to the
  appropriate FileSystem.  For example, if a FileSystem is mounted at
  /ppc2, an open operation for a file called /ppc2/profile.xml would be
  delegated to the mounted FileSystem. The mounted FileSystem will be
  given the filename relative to it. In this example the FileSystem's
  open operation would receive /profile.xml as the fileName argument.
```

```
Another example of this concept can be shown using the copy operation.
When a client invokes the copy operation, the FileManager will delegate
operations to the appropriate FileSystems (based upon supplied
pathnames) thereby allowing copy of files between filesystems.

If a client does not need to mount and unmount FileSystems, it can
treat the FileManager as a FileSystem by CORBA widening a FileManager
reference to a FileSystem reference.  One can always widen a
FileManager to a FileSystem since the FileManager is derived from a
FileSystem.

The FileManager's inherited FileSystem operations behavior implements
the FileSystem operations against the mounted file systems.  The
FileSystem operations ensure that the filename/directory arguments
given are absolute pathnames relative to a mounted FileSystem.  The
FileManager's FileSystem operations removes the FileSystem mounted name
from the input fileName before passing the fileName to an operation on
a mounted FileSystem.

The FileManager uses the mounted FileSystem for FileSystem operations
based upon the mounted FileSystem name that exactly matches the input
fileName to the lowest matching subdirectory.

The query operation returns the combined mounted file systems
information to the calling client based upon the given input
fileSystemProperties' IDs.  As a minimum, the query operation supports
the following input fileSystemProperties IDs:
1. SIZE - a property item ID value of "SIZE" will cause the query
operation to return the combined total size of all the mounted file
system as an unsigned long long property value.
2. AVAILABLE_SPACE - a property item ID value of "AVAILABLE_SPACE" will
cause the query operation to return the combined total available space
(in octets) of all the mounted file system as unsigned long long
property value.

The query operation raises the UnknownFileSystemProperties exception
when the input fileSystemProperties parameter contains an invalid
property ID. */

interface FileManager : FileSystem {
      /* The Mount structure identifies a FileSystem mounted within a
      FileManager. */

      struct MountType {
            FileSystem fs;
            string mountPoint;
      };

      /* This type defines an unbounded sequence of mounted
      FileSystems. */

      typedef sequence <MountType> MountSequence;

      /* This exception indicates a mount point does not exist within
      the FileManager */
```

```
exception NonExistentMount {
};

/* This exception indicates the FileSystem is a null (nil) object
reference. */

exception InvalidFileSystem {
};

/* This exception indicates the mount point is already in use in
the file manager. */

exception MountPointAlreadyExists {
};

/* The FileManager supports the notion of a federated file
system.  To create a federated file system, the mount operation
associated a FileSystem with a mount point (a directory name).

The mount operationassociates the specified FileSystem with the
given mountPoint.  A mountPoint name begins with a "/".  A
mountPoint name is a logical directory name for a FileSystem.

The mount operation raises the NonExistentMount exception when
the mountPoint (directory) name is not an absolute pathname
relative to the mounted file.

The mount operation raises the MountPointAlreadyExists exception
when the mountPoint already exists in the file manager.

The InvalidFileSystem exception is raised when the input
FileSystem is a null object reference.
@roseuid 36FAA01001E0 */
void mount (
      in string mountPoint,
      in FileSystem file_System
      )
      raises (InvalidFileName, InvalidFileSystem,
              MountPointAlreadyExists);

/* The unmount operation removes a mounted FileSystem from the
FileManager whose mounted name matches the input mountPoint name.
The unmount operation raises NonExistentMount when the mount
point does not exist within the FileManager.
@roseuid 36FAA07A010E */
void unmount (
      in string mountPoint
      )
      raises (NonExistentMount);

/* The getMounts operation returns the FileManager's mounted
FileSystems.
@roseuid 3895C9C3A0F0 */
MountSequence getMounts ();
```

```
};

/* This interface provides operations for managing associations between
ports.  An application defines a specific Port type by specifying an
interface that inherits the Port interface.  An application establishes
the operations for transferring data and control.  The application also
establishes the meaning of the data and control values.  Examples of
how applications may use ports in different ways include: push or pull,
synchronous or asynchronous, mono- or bi-directional, or whether to use
flow control (e.g., pause, start, stop).

The nature of Port fan-in, fan-out, or one-to-one is component
dependent.

Note 1:  The CORBA specification defines only a minimum size for each
basic IDL type.  The actual size of the data type is dependent on the
language (defined in the language mappings) as well as the Central
Processing Unit (CPU) architecture used.  By using these CORBA basic
data types, portability is maintained between components implemented in
differing CPU architectures and languages.

Note 2:  How components' ports are connected is described in the
software assembly descriptor file of the Domain Profile. */

interface Port {
      /* This exception indicates one of the following errors has
      occurred in the specification of a Port association:
      · errorCode 1 means the Port component is invalid (unable to
                    narrow object reference) or illegal object
                    reference,
      · errorCode 2 means the Port name is not found (not used by this
                    Port). */

      exception InvalidPort {
            string msg;
            unsigned short errorCode;
      };

      /* This exception indicates the Port is unable to accept any
      additional connections. */

      exception OccupiedPort {
      };

      /* The connectPort operation makes a connection to the component
      identified by the input parameters. The connectPort operation
      establishes only half of the association.

      A port may support several connections.  The input connectionID
      is a unique identifier to be used by disconnectPort when breaking
      this specific connection.

      The connectPort operation raises the InvalidPort exception when
      the input connection parameter is an invalid connection for this
      Port.  The OccupiedPort exception is raised when the Port is
      fully occupied and unable to accept connections.
      @roseuid 38C1759DA718 */
      void connectPort (
            in Object connection,
            in string connectionID
            )
            raises (InvalidPort, OccupiedPort);

      /* The disconnectPort operation breaks the connection to the
      component identified by the input parameters.

      The InvalidPort exception is raised when the name passed to the
      operation is invalid.
      @roseuid 38C175A5DC10 */
      void disconnectPort (
            in string connectionID
            )
            raises (InvalidPort);

};

/* The LifeCycle interface defines the generic operations for
initializing or releasing an instantiated component specific data
and/or processing elements. */

interface LifeCycle {
      /* This exception indicates an error occurred during component
      initialization. The messages provides additional information
      describing the reason why the error occurred. */

      exception InitializeError {
            StringSequence errorMessages;
      };

      /* This exception indicates an error occurred during component
      releaseObject. The messages provides additional information
      describing the reason why the errors occurred. */

      exception ReleaseError {
            StringSequence errorMessages;
      };

      /* The purpose of the initialize operation is to provide a
      mechanism to set an object to an known initial state.  (For
      example, data structures may be set to initial values, memory may
      be allocated, hardware devices may be configured to some state,
      etc.).

      Initialization behavior is implementation dependent.

      This operation raises the InitializeError when an initialization
      error occurs.
      @roseuid 37DD15FA01C2 */
      void initialize ()
            raises (InitializeError);

      /* The purpose of the releaseObject operation is to provide a
      means by which an instantiated component may be torn down. The
      releaseObject operation releases itself from the CORBA ORB.

      The releaseObject operation releases all internal memory
      allocated by the component during the life of the component. The
      releaseObject operation tears down the component (i.e. released
      from the CORBA environment).  The releaseObject operation
      releases components from the Operating Environment.

      This operation raises a ReleaseError when a release error occurs.
      @roseuid 37DD15FA01C3 */
      void releaseObject ()
            raises (ReleaseError);

};

/* The TestableObject interface defines a set of operations that can be
used to test component implementations. */

interface TestableObject {
      /* This exception indicates the requested testid for a test to be
      performed is not known by the component. */

      exception UnknownTest {
      };

      /* The runTest operation allows components to be "blackbox"
      tested.  This allows Built-In Test (BIT) to be implemented and
      this provides a means to isolate faults (both software and
      hardware) within the system.

      The runTest operation uses the testid parameter to determine
      which of its predefined test implementations should be performed.
      The testValues parameter Properties (id/value pair(s)) are used
      to provide additional information to the implementation-specific
      test to be run.  The runTest operation returns the result(s) of
      the test in the testValues parameter.

      Tests to be implemented by a component are component-dependent
      and are specified in the component's Properties Descriptor.
      Valid testid(s) and both input and ouput testValues (properties)
      for the runTest operation, at a minimum, are test properties
      defined in the properties test element of the component's
      Properties Descriptor (refer to Appendix D Domain Profile).  The
      testid parameter corresponds to the XML attribute testid of the
      property element test in a propertyfile.

      Before an UnknownProperties exception is raised by the runTest
      operation all inputValues properties are validated (i.e., test
      properties defined in the propertyfile(s) referenced in the
      component's SPD).

      The runTest operation does not execute any testing when the input
      testid or any of the the input testValues are not known by the
      component or are out of range.
```

```
        This operation does not return a value.

        The runTest operation raises the UnknownTest exception when there
        is no underlying test implementation that is associated with the
        input testid given.

        The runTest operation raises CF UnknownProperties exception when
        the input parameter testValues contains any DataTypes that are
        not known by the component's test implementation or any values
        that are out of range for the requested test. The exception
        parameter invalidProperties contains the invalid inputValues
        properties id(s) that are not known by the component or the
        value(s) are out of range.
        @roseuid 38A583C40208 */
        void runTest (
              in unsigned long testid,
              inout Properties testValues
              )
              raises (UnknownTest, UnknownProperties);

  };

  /* The PropertySet interface defines configure and query operations to
  access component properties/attributes. */

  interface PropertySet {
        /* This exception indicates the configuration of a component has
        failed (no configuration at all was done). The message provides
        additional information describing the reason why the error
        occurred. The invalid properties returned indicates the
        properties that were invalid. */

        exception InvalidConfiguration {
              Properties invalidProperties;
              string msg;
        };

        /* This exception indicates the configuration of a component was
        partially successful. The invalid properties returned indicates
        the properties that were invalid. */

        exception PartialConfiguration {
              Properties invalidProperties;
        };

        /* The purpose of this operation is to allow id/value pair
        configuration properties to be assigned to components
        implementing this interface.

        The configure operation shall assign values to the properties as
        indicated in the configProperties argument. An component's SPD
        profile indicates the valid configuration values.  Valid
        properties for the configure operation are at a minimum the
        configure readwrite and writeonly properties referenced in the
        component's SPD.
        The configure operation raises an InvalidConfiguration exception
        when a configuration error occurs that prevents any property
        configuration on the component.

        This operation raises PartialConfiguration exception when some
        configuration properties were successful and some configuration
        properties were not successful.
        @roseuid 38A583FFC998 */
        void configure (
              in Properties configProperties
              )
              raises (InvalidConfiguration, PartialConfiguration);

        /* The purpose of this operation is to allow a component to be
        queried to retrieve its properties.

        If the configProperties are zero size then, the query operation
        returns all component properties.  If the configProperties are
        not zero size, then the query operation returns only those
        id/value pairs specified in the configProperties. An component's
        SPD profile indicates the valid query types. Valid properties for
        the query operation are at a minimum the configure readwrite and
        readonly properties, and allocation properties that have an
        action value of "external" as referenced in the component's SPD.

        This operation raises the CF UnknownProperties exception when one
        or more properties being requested are not known by the
        component.
        @roseuid 38A583FFC99A */
        void query (
              inout Properties configProperties
              )
              raises (UnknownProperties);

  };

  /* The DomainManager interface API is for the control and configuration
  of the radio domain.

  The DomainManager interface can be logically grouped into three
  categories:
  Human Computer Interface (HCI), Registration, and Core Framework (CF)
  administration.

  1. The HCI operations are used to configure the domain, get the
  domain's capabilities (Devices, Services, and Applications), and
  initiate maintenance functions.  Host operations are performed by a
  client user interface capable of interfacing to the Domain Manager.

  2. The registration operations are used to register / unregister
  DeviceManagers, DeviceManager's Devices, DeviceManager's Services, and
  Applications at startup or during run-time for dynamic device, service,
  and application extraction and insertion.

  3. The administration operations are used to access the interfaces of
  registered DeviceManagers, FileManagers, and Loggers of the domain.
```

```
The DomainManager restores ApplicationFactories after startup for
applications that were previously installed by the DomainManager
installApplication operation.  The DomainManager adds the restored
ApplicationFactories to the DomainManager's applicationFactories
attribute.


A DomainManager implementation may log to 0 to many Log references
(uses ports).  The Logs utilized by the DomainManager implementation
shall be defined in the  Domain Manager Configuration Descriptor (DMD).
See Appendix D for further description of the DMD file.

Once a service specified in the DMD is successfully registered with the
DomainManager (via registerDeviceManager or registerService
operations), the DomainManager shall begin to use the service (e.g.
Log). */

interface DomainManager : PropertySet {
      /* This exception indicates an application installation has not
      completed correctly. */

      exception ApplicationInstallationError {
      };

      /* This type defines an unbounded sequence of Applications. */

      typedef sequence <Application> ApplicationSequence;

      /* This type defines an unbounded sequence of application
      factories. */

      typedef sequence <ApplicationFactory> ApplicationFactorySequence;

      /* This type defines an unbounded sequence of device managers. */

      typedef sequence <DeviceManager> DeviceManagerSequence;

      /* This exception indicates the application ID is invalid. */

      exception InvalidIdentifier {
      };

      /* The DeviceManagerNotRegistered exception indicates the
      registering Device's DeviceManager is not registered in the
      DomainManager. A Device's DeviceManager has to be registered
      prior to a Device registration to the DomainManager. */

      exception DeviceManagerNotRegistered {
      };

      /* The domainManagerProfile attribute contains the
      DomainManager's profile.  The readonly domainManagerProfile
      attribute contains either a profile element with a file reference
      to the DomainManager's  (DMD) profile or the XML for the
      DomainManager's (DMD) profile.  Files referenced within the
```

```
profile will have to be obtained from the DomainManager's
FileManager. */

readonly attribute string domainManagerProfile;
/* The deviceManagers attribute is read-only containing a
sequence of registered DeviceManagers in the domain.  The
DomainManager contains a list of registered DeviceManagers that
have registered with the DomainManager.  The DomainManager writes
an ADMINISTRATIVE_EVENT log to a DomainManager's Log, when the
deviceManagers attribute is obtained by a client. */

readonly attribute DeviceManagerSequence deviceManagers;
/* The applications attribute is read-only containing a sequence
of instantiated Applications in the domain.  The DomainManager
contains a list of Applications that have been instantiated. The
DomainManager writes an ADMINISTRATIVE_EVENT log record to a
DomainManager's Log, when the applications attribute is obtained
by a client. */

readonly attribute ApplicationSequence applications;
/* The readonly applicationFactories attribute contains a list
with one ApplicationFactory per application (SAD file and
associated files) successfully installed (i.e. no exception
raised).  The DomainManager writes an ADMINISTRATIVE_EVENT log
record to a DomainManager's Log, when the applicationFactories
attribute is obtained by a client. */

readonly attribute
ApplicationFactorySequence applicationFactories;

/* The fileMgr attribute is read only containing the mounted
FileSystems in the domain.  The DomainManager writes an
ADMINISTRATIVE_EVENT log record to a DomainManager's Log, when
the fileMgr attribute is obtained by a client. */

readonly attribute FileManager fileMgr;

/* The registerDevice operation verifies that the input
parameters, registeringDevice and registeredDeviceMgr, are not
nil CORBA component references.

The registerDevice operation adds the registeringDevice and the
registeringDevice's attributes  (e.g., identifier,
softwareProfile's allocation properties, etc.) to the
DomainManager, if it does not already exist.

The registerDevice operation associates the input
registeringDevice with the input registeredDeviceMgr in the
DomainManager when the input registeredDeviceMgr is a valid
registered DeviceManager in the DomainManager.

The registerDevice operation, upon successful device
registration, writes an ADMINISTRATIVE_EVENT log record to a
DomainManager's Log, to indicate that the device has successfully
registered with the DomainManager.
```

```
Upon unsuccessful device registration, the registerDevice
operation writes a FAILURE_ALARM log record to a DomainManager's
Log, when the InvalidProfile exception is raised to indicate that
the registeringDevice has an invalid profile.


Upon unsuccessful device registration, the registerDevice
operation logs a Failure_Alarm event with DomainManager's Logger
for the DeviceManagerNotRegistered exception to indicate that the
device that cannot be registered to the Device due to the
DeviceManager is not registered with the DomainManager.

Upon unsuccessful device registration, the registerDevice
operation writes a FAILURE_ALARM log record to a DomainManager's
Log,  indicating that the device could not register because the
DeviceManager is not registered with the DomainManager.

Upon unsuccessful device registration, the registerDevice
operation writes a FAILURE_ALARM log record to a DomainManager's
Log, because of an invalid reference input parameter.

The registerDevice operation raises the CF InvalidProfile
exception when:

1. The Device's SPD file and the SPD's referenced files do not
exist or cannot be processed due to the file not being compliant
with XML syntax, or

2. The Device's SPD does not reference allocation properties.

The registerDevice operation raises a DeviceManagerNotRegistered
exception when the input registeredDeviceMgr (not nil reference)
is not registered with the DomainManager.

The registerDevice operation raises the CF InvalidObjectReference
exception when input parameters registeringDevice or
registeredDeviceMgr contains an invalid reference.
@roseuid 364B4CF92ED0 */
void registerDevice (
      in Device registeringDevice,
      in DeviceManager registeredDeviceMgr
      )
      raises (InvalidObjectReference, InvalidProfile,
              DeviceManagerNotRegistered);

/* This operation is used to register a DeviceManager, its
Device(s), and its Services.  Software profiles can also be
obtained from the DeviceManager's FileSystem.

The registerDeviceManager operation verifies that the input
parameter, deviceMgr, is a not a nil CORBA component reference.

The registerDeviceManager operation adds the input deviceMgr's
registeredServices and each registeredService's names to the
DomainManager.  The registerDeviceManager operation associates
the input deviceMgr's with the input deviceMgr's
```

```
registeredServices in the DomainManager in order to support the
unregisterDeviceManager operation.

The registerDeviceManager operation performs the documented
connections as specified in the connections element of the
deviceMgr's DCD file. For connections established for a Log, the
registerDeviceManager operation creates a unique producer log ID
for each log producer. The registerDeviceManager operation
invoke the PropertySet configure operation on each log producer
in order to set its unique PRODUCER_LOG_ID.  If the
DeviceManager's DCD describes a connection for a service that has
not been registered with the DomainManager, the
registerDeviceManager operation establishes any pending
connection when the service registers with the DomainManager by
the registerDeviceManager operation.

The registerDeviceManager operation adds the input deviceMgr to
the DomainManager's deviceManagers attribute, if it does not
already exist.

The registerDeviceManager operation adds the input deviceMgr's
registeredDevices and each registeredDevice's attributes  (e.g.,
identifier, softwareProfile's allocation properties, etc.) to the
DomainManager.

The registerDeviceManager operation associates the input
deviceMgr with the input deviceMgr's registeredDevices in the
DomainManager in order to support the unregisterDeviceManager
operation.

The registerDeviceManager operation obtains all the Software
profiles from the registering DeviceManager's FileSystems.

The registerDeviceManager operation mounts the DeviceManager's
FileSystem to the DomainManager's FileManager.  The mounted
FileSystem name shall have the format, "/DomainName/HostName",
where DomainName is the name of the domain and HostName is the
input deviceMgr's label attribute

The registerDeviceManager operation, upon unsuccessful
DeviceManager registration, writes a FAILURE_ALARM log record to
a DomainManager's Log.

The registerDeviceManager operation raises the CF
InvalidObjectReference exception when the input parameter
deviceMgr contains an invalid reference to a DeviceManager
interface.
@roseuid 364B4D632938 */
void registerDeviceManager (
      in DeviceManager deviceMgr
      )
      raises (InvalidObjectReference, InvalidProfile);

/* This operation is used to unregister a DeviceManager object
from the DomainManager's Domain Profile.
```

```
The unregisterDeviceManager operation unregisters a DeviceManager
component from the DomainManager.

The unregisterDeviceManager operation releases (client-side CORBA
release) all device(s) and service(s) associated with the
DeviceManager that is being unregistered.

The unregisterDeviceManager operation shall unmount all
DeviceManager's FileSystems from its File Manager.

The unregisterDeviceManager operation, upon the successful
unregistration of a DeviceManager, writes an ADMINISTRATIVE_EVENT
log record to a DomainManager's Log.

The unregisterDeviceManager operation, upon unsuccessful
unregistration of a DeviceManager, writes a FAILURE_ALARM log
record to a DomainManager's Log.


The unregisterDeviceManager operation raises the CF
InvalidObjectReference when the input parameter DeviceManager
contains an invalid reference to a DeviceManager interface.
@roseuid 364B4EAB15E0 */
void unregisterDeviceManager (
      in DeviceManager deviceMgr
      )
      raises (InvalidObjectReference);

/* The unregisterDevice operation removes a device entry from the
DomainManager's Domain Profile..

The unregisterDevice operation releases (client-side CORBA
release) the unregistering Device from the DomainManager.

The unregisterDevice operation, upon the successful
unregistration of a Device, writes an ADMINISTRATIVE_EVENT log
record to a DomainManager's Log.

The unregisterDevice operation, upon unsuccessful unregistration
of a Device, writes a FAILURE_ALARM log record to a
DomainManager's Log.

The unregisterDevice operation raises the CF
InvalidObjectReference exception when the input parameter
contains an invalid reference to a Device interface.
@roseuid 364B4EC8DEC0 */
void unregisterDevice (
      in Device unregisteringDevice
      )
      raises (InvalidObjectReference);

/* This operation is used to register new application software in
the DomainManager's Domain Profile. An installer application
typically invokes this operation when it has completed the
installation of a new Application into the domain.
```

```
The profileFileName is the absolute path of the profile filename.

The installApplication operation verifies the application's SAD
file exists in the DomainManager's FileManager and all the files
the application is dependent on are also resident.


The installApplication operation writes an ADMINISTRATIVE_EVENT
log Record to a DomainManager's Log, upon successful Application
installation.


The installApplication operation raises the
ApplicationInstallationError exception when the installation of
the Application file(s) was not successfully completed.

The installApplication operation raises the CF InvalidFileName
exception when the input SAD file or any referenced file name
does not exist in the file system as defined in the absolute path
of the input profileFileName.  The installApplication operation
logs a FAILURE_ALARM log record to a DomainManger's Log when the
InvalidFileName exception occurs and the logged message shall be
"installApplication:: invalid file is xxx", where "xxx" is the
input or referenced file name is bad.

The installApplication operation raises the CF InvalidProfile
exception when the input SAD file or any referenced file is not
compliant with XML DTDs defined in Appendix D or referenced
property definitions are missing.  The installApplication
operation s logs a FAILURE_ALRAM log record ot a DomainManager's
Log when the CF InvalidProfile exception occurs and the logged
message shall be "installApplication:: invalid Profile is yyy,"
where "yyy" is and the input or referenced file name that is bad
along with the element or position within the profile that is
bad.
@roseuid 3896F0D83588 */
void installApplication (
      in string profileFileName
      )
      raises (InvalidProfile, InvalidFileName,
              ApplicationInstallationError);

/* This operation is used to uninstall an application in the
DomainManager's Domain Profile.  The CF Installer typically
invokes this operation when removing an application from the
radio domain.

The uninstallApplication operation removes all files associated
with the Application.

The uninstallApplication operation makes the ApplicationFactory
unavailable from the DomainManager (i.e. its services no longer
provided for the Application).

The uninstallApplication operation, upon successful uninstall of
an Application, writes an ADMINISTRATIVE_EVENT log record to a
```

```
DomainManager's Log.

The uninstallApplication operation, upon unsuccessful uninstall
of an  Application, writes a FAILURE_ALARM log record to a
DomainManager's Log.

The uninstallApplication operation  raises the InvalidIdentifier
exception when the ApplicationID is invalid.
@roseuid 3896F13747C8 */
void uninstallApplication (
      in string applicationID
      )
      raises (InvalidIdentifier);

/* This operation is used to register a service for a specific
DeviceManager with the DomainManager.

The registerService operation verifies the input
registeringService and registeredDeviceMgr are valid object
references .The registerService operation verifies the input
registeredDeviceMgr has been previously registered with the
DomainManager. The registerService operation adds the
registeringService and the registeringService's name to the
DomainManager.  However, if the name of the registering service
is not a unique name for that type of service (i.e. it is a
duplicate name of an already registered service of the same type
of service), then the new service is not be registered by the
DomainManager.The registerService operation associates the input
registeringService with the input registeredDeviceMgr in the
DomainManager when the input registeredDeviceMgr is a valid
registered DeviceManager in the DomainManager. The
registerService operation, upon successful service registration,
establish any pending connection requests for the
registeringService. For connections established for a Log, the
registerService operation creates a unique producer log ID for
each log producer.  The registerService operation  invokes the
PropertySet configure operation onceon each log producer in order
to set its unique PRODUCER_LOG_ID (see section 3.1.3.3.5.5.1.2
for details).

The registerService operation, upon successful service
registration, writes a log record to a DomainManager's Log, with
the log level set to ADMINISTRATIVE_EVENT.

The registerService operation, upon unsuccessful service
registration, writes a log record to a DomainManager's Log, with
the log level set to FAILURE_ALARM.

This operation does not return a value.

The registerService operation raises a DeviceManagerNotRegistered
exception when the input registeredDeviceMgr (not nil reference)
is not registered with the DomainManager. The registerService
operation raises the CF InvalidObjectReference exception when
input parameters registeringService or registeredDeviceMgr
contains an invalid reference.
```

```
      @roseuid 3B33926D032F */
      void registerService (
            in Object registeringService,
            in DeviceManager registeredDeviceMgr,
            in string name
            )
            raises (InvalidObjectReference, InvalidProfile,
                    DeviceManagerNotRegistered);

      /* This operation is used to remove a service entry from the
      DomainManager for a specific DeviceManager.


      The unregisterService operation removes a service entry from the
      DomainManager. The unregisterService operation releases (client-
      side CORBA release) the unregisteringService from the
      DomainManager.  The unregisterService operation, upon the
      successful unregistration of a Service, writes a log record to a
      DomainManager's Log, with the log level set to
      ADMINISTRATIVE_EVENT.  The unregisterService operation, upon
      unsuccessful unregistration of a Service, write a log record to a
      DomainManager's Log, with the log level set to FAILURE_ALARM.

      This operation does not return a value.

      The unregisterService operation raises the CF
      InvalidObjectReference exception when the input parameter
      contains an invalid reference to a Service interface.
      @roseuid 3B3392750114 */
      void unregisterService (
            in Object unregisteringService,
            in string name
            )
            raises (InvalidObjectReference);

};

/* The ApplicationFactory interface class provides an interface to
request the creation of a specific type  (e.g., SINCGARS, LOS,
Havequick, etc.) of Application in the domain. The ApplicationFactory
interface class is designed using the Factory Design Pattern.  The
Software Profile determines the type of Application that is created by
the ApplicationFactory. */

interface ApplicationFactory {
      /* This exception is raised when the parameter
      DeviceAssignmentSequence contains one (1) or more invalid
      Application component-to-device assignment(s). */

      exception CreateApplicationRequestError {
            DeviceAssignmentSequence invalidAssignments;
      };

      /* This exception is raised when the create request is valid but
      the Application is unsuccessfully instantiated due to internal
      processing errors. */
```

```
exception CreateApplicationError {
      StringSequence errorMessages;
};

/* The invalidInitConfiguration exception is raised when the
input initConfiguration parameter is invalid. */

exception InvalidInitConfiguration {
      Properties invalidProperties;
};

/* The name attribute contains the name of the type of
Application that can be instantiated by the ApplicationFactory
(e.g., SINCGARS, LOS, Havequick, DAMA25, etc.). */

readonly attribute string name;
/* This attribute contains the application software profile that
this factory uses when creating an application.  The string value
contains either a profile element with a file reference to the
SAD profile file or the actual xml for the SAD profile.  Files
referenced within the profile will have to be obtained from a
FileManager.  The ApplicationFactory will have to be queried for
profile information for Component files that are referenced by an
ID instead of file name. */

readonly attribute string softwareProfile;

/* This operation is used to create an Application within the
system domain.

The create operation provides a client interface to request the
creation of an Application on client requested device(s) or the
creation of an Application in which the ApplicationFactory
determines the necessary device(s) required for instantiation of
the Application.

If the input parameter CF DeviceAssignmentsSequence length is
zero (0), the ApplicationFactory allocates the devices for the
creation of the Application as specified in the Software
Profile's software assembly descriptor (SAD).  Each application
will have a SAD.

An Application can be comprised of one or more components (e.g.,
Resources, Devices, etc.).  The SAD contains Software Package
Descriptors (SPDs) for each Application component.  The SPD
specifies the Device implementation criteria for loading
dependencies (processor kind, etc.) and processing capacities
(e.g., memory, process) for an application component.  The create
operation uses the SAD SPD implementation element to locate
candidate devices capable of loading and executing Application
components.

If deviceAssignments (not zero length) are provided, the
ApplicationFactory verifies each device assignment, for the
specified component, against the component's SPD implementation
```

element.

The create operation allocates (Device::allocateCapacity) component capacity requirements against candidate devices to determine which candidate devices satisfy all SPD implementation criteria requirements and SAD partitioning requirements (e.g, components HostCollocation, etc.). The create operation only uses devices that have been granted successful capacity allocations for loading and executing Application components, or used for data processing. The actual devices chosen will reflect changes in capacity based upon component capacity requirements allocated to them, which may also cause state changes for the Devices.

The create operation loads the Application components (including all of the Application-dependent components) to the chosen device(s).

The create operation executes the Application components (including all of the Application-dependent components) using the entry points dictated in the SPD's implementation code element. paragraph The create operation uses the component's SPD implementation code's stacksize and priority elements, when specified,for the execute options parameters. Parameters passed to entry points will be in the form of a naming context parameter with an ID of "NAMING_CONTEXT" and string value of (/ DomainName / NodeName / [other context sequences]) / ComponentName_UniqueIdentifier, when the component object reference is to be retrieved from a Naming Service as indicated by the SAD." The unique identifier is determined by the implementation, unique to each node. The create operation uses this naming string to form component names that need to be retrieved from Naming Service. When the NAMING_CONEXT parameter is used, the create operation forms a naming service IOR with the format of an ID of "NAMING_SERVICE_IOR" and string value of the CORBA Naming Service IOR that the ApplicationFactory is using. Due to the dynamics of bind and resolve to Naming Service, the create operation should provide sufficient attempts to retrieve component object references from Naming Service prior to generating an exception.

In the naming parameter string, each "slash" (/) represents a separate naming context. A naming context is made up of ID and kind pair, which is indicated by "id.kind" in the naming parameter string. The naming context kind is optional and when not specified a null string will be used.

The NodeName naming context ID value is the label of the DeviceManager in which the component was loaded and going to be executed.

The ComponentName naming context ID value is the component instantiation findcomponent findby namingservice name element value in the software assembly descriptor (SAD).

The create operation passes the componentinstantiation element "execparam" properties that have values as parameters to execute operation. The create operation passes "execparam" parameters values as string values.

The create operation, initializes Resources first, then establishes connections for Resources, and finally configures the Resources.

The create operation will only configure the application's assemblycontroller component. The create operation initializes an Application component provided the component implements the LifeCycle interface.

The create operation configures an application's assemblycontroller component provided the assemblycontroller has configure readwrite or writeonly properties with values. The create operation uses the union of the input initConfiguration properties of the create operation and the assemblycontroller's componentinstantiation writeable "configure" properties that have values. The input initConfiguration parameter have precedences over the assemblycontroller's writeable "configure" property values.

The create operation, when creating a component from a CF ResourceFactory, passes the componentinstantiation componentresoursefactoryref element "factoryparam" properties that have values as qualifiers parameters to the referenced ResourceFactory component's createResource operation.

The create operation interconnects Application components' (Resources' or Devices') ports in accordance with the SAD. The create operation obtain Ports in accordance with the SAD via Resource getPort operation.The create operation uses the SAD connectinterface element id attribute as the unique identifier for a specific connection when provided. The create operation creates a connection ID when no SAD connectinterface element attribute id is specified for a connection. The create operation obtains a Resource in accordance with the SAD via the CORBA Naming Service, ResourceFactory, or a stringified IOR. The ResourceFactory can be obtained by using the CORBA Naming Service or a stringified IOR as stated in the SAD.

The create operation passes, with invocation of each ResourceFactory createResource operation, the ResourceFactory configuration properties associated with that Resource as dictated by the SAD.

The dependencies to Logger and FileManager will show up as connections in the SAD.

If the Application is successfully created, the create operaiton returns an Application component reference for the created Application. A sequence of created Application references can be obtained using the DomainManager::getApplications operation.

```
        The create operation, upon successful Application creation,
        writes an ADMINISTRATIVE_EVENT log record to a DomainManager's
        Log.

        The create operation, upon unsuccessful Application creation,
        writes a FAILURE_ALARM log record to a DomainManager's Log.

        The create operation raises the CreateApplicationRequestError
        exception when the parameter CF DeviceAssignmentSequence contains
        one (1) or more invalid Application component to device
        assignment(s).

        The create operation raises the CreateApplicationError exception
        when the create request is valid but the Application can not be
        successfully instantiated due to internal processing error(s).

        The create operation raises the InvalidInitConfiguration
        exception when the input initConfiguration parameter is invalid.
        The InvalidInitConfiguration invalidPropertiesl identifies the
        property that is invalid.
        @roseuid 38B7D97BCF98 */
        Application create (
              in string name,
              in Properties initConfiguration,
              in DeviceAssignmentSequence deviceAssignments
              )
              raises (CreateApplicationError,
                      CreateApplicationRequestError,
                      InvalidInitConfiguration);

  };

  /* This interface provides the getPort operation for those objects that
  provide ports. */

  interface PortSupplier {
        /* This exception is raised if an undefined port is requested. */

        exception UnknownPort {
        };

        /* The getPort operation provides a mechanism to obtain a
        specific consumer or producer Port.  A PortSupplier may contain
        zero-to-many consumer and producer port components.  The exact
        number is specified in the component's Software Profile SCD
        (section Error! Reference source not found.).  These Ports can be
        either push or pull types.  Multiple input and/or output ports
        provide flexibility for PortSuppliers that must manage varying
        priority levels and categories of incoming and outgoing messages,
        provide multi-threaded message handling, or other special message
        processing.

        The getPort operations returns the object reference to the named
        port as stated in the component's SCD.  The getPort operation
        returns the CORBA object reference that is associated with the
        input port name.
```

```
      The getPort operation raises an UnknownPort exception if the port
      name is invalid.
      @roseuid 3B336BB80213 */
      Object getPort (
            in string name
            )
            raises (UnknownPort);

};

/* The Resource interface provides a common API for the control and
configuration of a software component.  The Resource interface inherits
from the LifeCycle, PropertySet, TestableObject, and PortSupplier
interfaces.

The inherited LifeCycle, PropertySet, TestableObject, and PortSupplier
interface operations are documented in their respective sections of
this document.

The Resource interface may also be inherited by other application
interfaces as described in the Software Profile's Software Component
Descriptor (SCD) file. */

interface Resource : LifeCycle, TestableObject, PropertySet,
                     PortSupplier {
      /* This exception indicates a Start error has occurred for the
      Resource.  An error message is given explainng the start error.
      */

      exception StartError {
            string msg;
      };

      /* This exception indicates a Stop error has occurred for the
      Resource.  An error message is given explainng the stop error. */

      exception StopError {
            string msg;
      };

      /* The start operation puts the Resource in an operating
      condition.

      The start operation raises the StartError exception if an error
      occurs while starting the resource.
      @roseuid 38BEE2457548 */
      void start ()
            raises (StartError);

      /* The stop operation disables all current operations and put the
      Resource in a non-operating condition.  Subsequent configure,
      query, and start operations are not inhibited by the stop
      operation.

      The stop operation raises the StopError exception if an error
      occurs while stopping the resource.
      @roseuid 38BEE2457549 */
      void stop ()
            raises (StopError);

};

/* A Device is a type of Resource within the domain and has the
requirements as stated in the Resource interface.  This interface
defines additional capabilities and attributes for any logical Device
in the domain.  A logical Device is a functional abstraction for a set
(e.g., zero or more) of hardware devices and provides the following
attributes and operations:

1. Software Profile Attribute - This SPD XML profile defines the
logical Device capabilities (data/command uses and provides ports,
configure and query properties, capacity properties, status properties,
etc.), which could be a subset of the hardware device's capabilities.

2. State Management Attributes - The usage, operational, and
administrative states constitutes the overall state for a logical
Device. Status properties may contain more detailed information about
aspects of the states.

3. Capacity Operations - In order to use a device, certain capacities
(e.g., memory, performance, etc.) must be obtained from the Device.
The capacity properties will vary among devices and are described in
the Software Profile. A device may have multiple allocatable
capacities, each having its own unique capacity model.

The following behavior is in addition to the LifeCycle releaseObject
operation behavior.

The releaseObject operation calls the releaseObject operation on all of
the Device's aggregated Devices (i.e., those Devices that are contained
within the AggregateDevice'sdevices attribute).

The releaseObject operation transitions the Device's adminState to
SHUTTING_DOWNstate, when the Device's adminState is UNLOCKED.

The releaseObject operation causes the Device to be unavailable (i.e.
released from the CORBA environment, and process terminated on the OS
when applicable), when the Device's adminState transitions to LOCKED,
meaning its aggregated Devices have been removed and the Device's
usageState is IDLE.

The releaseObject operation shall cause the removal of its Device from
the Device's compositeDevice.  The releaseObject operation unregisters
its Device from its DeviceManager. */

interface Device : Resource {
      /* This exception indicates that the device is not capable of the
      behavior being attempted due to the state the Device is in.  An
      example of such behavior is allocateCapacity.
      exception InvalidState {string msg;}; */
      exception InvalidState {
            string msg;
      };

      /* The InvalidCapacity exception returns the capacities that are
      not valid for this device.exception InvalidCapacity. */

      exception InvalidCapacity {
            /* The invalid capacities sent to the allocateCapacity
            operation. */
            Properties capacities;
            /* The message indicates the reason for the invalid
            capacity. */
            string msg;
      };

      /* This is a CORBA IDL enumeration type that defines a Device's
      administrative states.  The administrative state indicates the
      permission to use or prohibition against using the Device. */

      enum AdminType {
            UNLOCKED,
            LOCKED,
            SHUTTING_DOWN
      };

      /* This is a CORBA IDL enumeration type that defines a Device's
      operational states.  The operational state indicates whether or
      not the object is functioning. */

      enum OperationalType {
            ENABLED,
            DISABLED
      };

      /* This is a CORBA IDL enumeration type that defines the Devices
      usage states.  This state indicates whether or not a Device is:
      IDLE        - not in use
      ACTIVE      - in use, with capacity remaining for allocation or
      BUSY        - in use, with no capacity remaining for allocation
      */

      enum UsageType {
            IDLE,
            ACTIVE,
            BUSY
      };

      /* The readonly compositeDevice attribute contains the object
      reference of the AggregateDevice with which this Device is
      associated or a nil CORBA object reference if no association
      exists. */

      readonly attribute AggregateDevice compositeDevice;
      /* The readonly usageState attribute contains the Device's usage
      state (IDLE, ACTIVE, or BUSY).  UsageState indicates whether or
```

```
not a device is actively in use at a specific instant, and if so,
whether or not it has spare capacity for allocation at that
instant. */

readonly attribute UsageType usageState;
/* The administrative state indicates the permission to use or
prohibition against using the device.  The adminState attribute
contains the device's admin state value.  The adminState
attribute only allows the setting of LOCKED and UNLOCKED values,
where "LOCKED" is only effective when the adminState attribute
value is UNLOCKED, and "UNLOCKED" is only effective when the
adminState attribute value is LOCKED or SHUTTING_DOWN.  Illegal
state transitions commands are ignored.The adminState, upon being
commanded to be LOCKED,  transitions from the UNLOCKED to the
SHUTTING_DOWN state and set the adminState to LOCKED for its
entire aggregation of Devices (if it has any).  The adminState
then transitions to the LOCKED state when the Device's usageState
is IDLE and its entire aggregation of  Devices are LOCKED. */

attribute AdminType adminState;
/* The readonly label attribute contains the Device's label.  The
label attribute is the meaningful name given to a Device.  The
attribute could convey location information within the system
(e.g., audio1, serial1, etc.). */

readonly attribute string label;
/* The softwareProfile attribute is the XML description for this
logical Device.The readonly softwareProfile attribute contains
either a profile DTD element with a file reference to the SPD
profile file or the XML for the SPD profile.  Files referenced
within the softwareProfile are obtained via the FileManager. */

readonly attribute string softwareProfile;
/* The readonly identifier attribute contains the unique
identifier for a device instance. */

readonly attribute string identifier;
/* The readonly operationalState attribute contains the device's
operational state (ENABLED or DISABLED).  The operational state
indicates whether or not the device is functioning. */

readonly attribute OperationalType operationalState;

/* This operation provides the mechanism to request and allocate
capacity from the Device.

The allocateCapacity operation reduces the capacities of the
Device based upon the capacities requested, when the adminState
is UNLOCKED, operationalState is ENABLED, and usageState is not
BUSY.

The allocateCapacity operation sets the Device's usageState
attribute to BUSY, when the Device determines that it is not
possible to allocate any further capacity.  The allocateCapacity
operation sets the usageState attribute to ACTIVE, when capacity
is being used and any capacity is still available for allocation.
```

```
        The allocateCapacity operation returns true, if the capacity has
        been allocated, or false if not allocated.

        The allocateCapacity operation raises the InvalidCapacity
        exception, when the capacities are invalid or the capacity values
        are the wrong type or ID.

        The allocateCapacity operation raises the InvalidState exception,
        when the Device's adminState is not UNLOCKED or operationalState
        is DISABLED.
        @roseuid 38B7EFD077B0 */
        boolean allocateCapacity (
              in Properties capacities
              )
              raises (InvalidCapacity, InvalidState);

        /* This operation provides the mechanism to return capacities
        back to the Device, making them available to other users.

        The deallocateCapacity operation adjusts the capacities of the
        Device based upon the input capacities parameter.

        The deallocateCapacity operation sets the usageState attribute to
        ACTIVE when, after adjusting capacities, any of the Device's
        capacities are still being used.

        The deallocateCapacity operation sets the usageState attribute to
        IDLE when, after adjusting capacities, none of the Device's
        capacities are still being used.

        The deallocateCapacity operation sets the adminState attribute to
        LOCKED as specified in adminState attribute.

        This operation does not return any value.

        The deallocateCapacity operation raises the InvalidCapacity
        exception, when the capacity ID is invalid or the capacity value
        is the wrong type.  The InvalidCapacity exception will state the
        reason for the exception.

        The deallocateCapacity operation raises the InvalidState
        exception, when the Device's adminState is LOCKED or
        operationalState is DISABLED.
        @roseuid 38B7EFFDDD48 */
        void deallocateCapacity (
              in Properties capacities
              )
              raises (InvalidCapacity, InvalidState);

  };

  /* The Application provides the interface for the control,
  configuration, and status of an instantiated application in the domain.
  The Application interface inherits the IDL interface of CF Resource.
  A created application instance may contain CF Resource components
```

and/or non-CORBA components.

The Application interface releaseObject operation provides the interface to release the computing resources allocated during the instantiation of the Application, and de-allocate the devices associated with Application instance. An instance of an Application is returned by the create operation of an instance of the ApplicationFactory.

The Application delegates the implementation of the inherited Resource operations (runTest, start, stop, configure, and query) to the application's assemblycontroller Resource. The Application propagates exceptions raised by the application's assemblycontroller's operations. The initialize operation is not propagated to Application components or the assemblycontroller, and causes no action within an Application.

releaseObject Behavior
For each Application component not created by a ResourceFactory, the releaseObject operation releases the component by utilizing the Resources's releaseObject operation. If the component was created by a ResourceFactory, the releaseObject operation releases the component by the ResourceFactory releaseResource operation. The releaseObject operation shutdowns a ResourceFactory when no more Resources are managed by the ResourceFactory.

For each allocated device capable of operation execution, the releaseObject operation terminates all processes / tasks of the Application's components utilizing the Device's terminate operation.

For each allocated device capable of memory function, the releaseObject operation de-allocates the memory associated with Application's component instances utilizing the Device's unload operation.

The releaseObject operation deallocates the Devices that are associated with the Application being released, based on the Application's Software Profile.

The actual devices deallocated (Device::deallocateCapacity) will reflect changes in capacity based upon component capacity requirements deallocated from them, which may also cause state changes for the Devices.

The Application releases all client component references to the Application components.

The releaseObject operation disconnects Ports from other Ports that have been connected based upon the software profile.

For components (e.g., Resource, ResourceFactory) that are registered with Naming Service, the releaseObject operation unregisters those components from Naming Service.

The releaseObject operation for an application disconnects Ports first, then release the Resources and ResourceFactories, then call the terminate operation, and lastly call the unload operation on the

```
devices.

The releaseObject operation, upon successful Application release,
writes an ADMINISTRATIVE_EVENT log record to a DomainManager's Log.

The releaseObject operation, upon unsuccessful Application release,
writes a FAILURE_ALARM log record to a DomainManager's Log. */

interface Application : Resource {
      /* The ComponentProcessIdType defines a type for associating a
      component with its process ID.  This type can be used to retrieve
      a process ID for a specific component. */

      struct ComponentProcessIdType {
            /* The componentID is a ID of a component that corresponds
            to the application's SAD componentinstantiation's ID
            attribute value. */
            string componentID;
            /* The process ID of the executable component. */
            unsigned long processId;
      };

      /* The ComponentProcessIdSequence type defines an unbounded
      sequence of components' process IDs. */

      typedef
      sequence <ComponentProcessIdType> ComponentProcessIdSequence;

      /* The ComponentElementType defines a type for associating a
      component with an element (e.g., naming context, implementation
      ID). */

      struct ComponentElementType {
            /* The componentID is a ID of a component that corresponds
            to the application's SAD componentinstantiation's ID
            attribute value. */
            string componentID;
            /* The element ID that is associated with component ID. */
            string elementID;
      };

      /* The ComponentElementSequence defines an unbounded sequence of
      components with an associated element. */

      typedef sequence <ComponentElementType> ComponentElementSequence;

      /* The componentNamingContexts attribute contains the list of
      components' Naming Service Context within the Application for
      those components using CORBA Naming Service. */

      readonly attribute
      ComponentElementSequence componentNamingContexts;
      /* The componentProcessIds attribute contains the list of
      components' process IDs within the Application for components
      that are executing on a device. */
```

```
	readonly attribute
	ComponentProcessIdSequence componentProcessIds;
	/* The componentDevices attribute shall contains the list of
	components' device assignments within the application.  Each
	component (componentinstantiation element in the Application's
	SAD) is associated with a device. */

	readonly attribute DeviceAssignmentSequence componentDevices;
	/* The componentImplementations attribute contains the list of
	components' SPD implementation IDs within the Application for
	those components created. */

	readonly attribute
	ComponentElementSequence componentImplementations;
	/* This attribute is the XML profile information for the
	application.  The string value contains either a profile element
	with a file reference to the SAD profile file or the actual xml
	for the SAD profile.  Files referenced within a profile will have
	to be obtained from a FileManager.  The Application will have to
	be queried for profile information for Component files that are
	referenced by an ID instead of a file name. */

	readonly attribute string profile;
	/* This name attribute contains the name of the created
	Application.  The ApplicationFactory interface's create operation
	name parameter provides the name content. */

	readonly attribute string name;
};

/* This interface extends the Device interface by adding software
loading and unloading behavior to a Device. */

interface LoadableDevice : Device {
	/* This type defines the type of load to be performed. The load
	types are in accordance with Appendix D SPD code element. */

	enum LoadType {
		DRIVER,
		KERNEL_MODULE,
		SHARED_LIBRARY,
		EXECUTABLE
	};

	/* This exception indicates that the device is unable to load
	that type of file, as identified by the loadKind parameter. */

	exception InvalidLoadKind {
	};

	/* This operation provides the mechanism for loading software on
	a specific device.  The loaded software may be subsequently
	executed on the Device, if the Device is an ExecutableDevice.

	The load operation loads a file on the specified device based
	upon the input loadKind and fileName parameters using the input
```

```
      FileSystem parameter to retrieve the file.

      The load operation supports the load types as stated in the
      Device's software profile LoadType allocation properties.

      The load operation keeps track of the number of times a file has
      been successfully loaded.

      This operation does not return any value.

      The load operation raises the InvalidState exception when the
      Device's adminState is not UNLOCKED or operationalState is
      DISABLED.

      The load operation raises the InvalidLoadKind exception when the
      input loadKind parameter is not supported.

      The load operation raises the CF InvalidFileName exception when
      the file designated by the input filename parameter cannot be
      found.
      @roseuid 3A5DAED301AE */
      void load (
            in FileSystem fs,
            in string fileName,
            in LoadType loadKind
            )
            raises (InvalidState, InvalidLoadKind, InvalidFileName);

      /* This operation provides the mechanism to unload software that
      is currently loaded.

      The unload operation decrements the load count for the input
      filename parameter by one. The unload operation unloads the
      application software on the device based on the input fileName
      parameter, when the file's load count equals zero.

      This operation does not return a value.

      The unload operation raises the InvalidState exception when the
      Device's adminState is LOCKED or its operationalState is
      DISABLED.

      The unload operation raises the CF InvalidFileName exception when
      the file designated by the input filename parameter cannot be
      found.
      @roseuid 3A5DAED301B2 */
      void unload (
            in string fileName
            )
            raises (InvalidState, InvalidFileName);

};

/* This interface extends the LoadableDevice interface by adding
execute and terminate behavior to a Device. */
```

```
interface ExecutableDevice : LoadableDevice {
      /* This exception indicates that a process, as identified by the
      processID parameter, does not exist on this device. */

      exception InvalidProcess {
      };

      /* This exception indicates that a function, as identified by the
      input name parameter, hasn't been loaded on this device. */

      exception InvalidFunction {
      };

      /* This type defines a process number within the system.  Process
      number is unique to the Processor operating system that created
      the process. */

      typedef unsigned long ProcessID_Type;

      /* The InvalidParameters exception indicates the input parameters
      are invalid on the execute operation. This exception is raised
      when there are invalid execute parameters. Each parameter's ID
      and value must be a valid string type. The invalidParms is a list
      of invalid parameters specified in the execute operation. */

      exception InvalidParameters {
            /* The invalidParms is a list of invalid parameters
            specified in the execute or executeProcess operation.  Each
            parameter's ID and value must be a string type. */
            Properties invalidParms;
      };

      /* The InvalidOptions exception indicates the input options are
      invalid on the execute operation. The invalidOpts is a list of
      invalid options specified in the execute operation. */

      exception InvalidOptions {
            /* The invalidParms is a list of invalid parameters
            specified in the execute or executeProcess operation.  Each
            parameter's ID and value must be a string type. */
            Properties invalidOpts;
      };

      /* The STACK_SIZE_ID is the identifier for the ExecutableDevice's
      execute options parameter.  The value for a stack size is an
      unsigned long. */

      const string STACK_SIZE = "STACK_SIZE";
      /* The PRIORITY_ID is the identifier for the ExecutableDevice's
      execute options parameters.  The value for a priority is an
      unsigned long. */

      const string PRIORITY_ID = "PRIORITY";
      /* The terminate operation provides the mechanism for terminating
      the execution of a process/thread on a specific device that was
      started up with the execute operation.
```

```
The terminate operation terminates the execution of the
process/thread designated by the processId input parameter on the
Device.

This operation does not return a value.

The terminate operation raises the InvalidState exception when
the Device's adminState is LOCKED or operationalState is
DISABLED.

The terminate operation raises the InvalidProcess exception when
the processId does not exist for the Device.
@roseuid 3A5DAEC1016D */
void terminate (
      in ProcessID_Type processId
      )
      raises (InvalidProcess, InvalidState);

/* This operation provides the mechanism for starting up and
executing a software process/thread on a device.

The execute operation executes the function or file identified by
the input name parameter using the input parameters and options
parameters. Whether the input name parameter is a function or a
file name is device-implementation-specific.The execute operation
converts the input parameters (id/value string pairs) parameter
to the standard argv of the POSIX exec family of functions, where
argv(0) is the function name. The execute operation maps the
input parameters parameter to argv starting at index 1 as
follows, argv (1) maps to input parameters (0) id and argv (2)
maps to input parameters (0) value and so forth. The execute
operation passes argv through the operating system "execute"
function.

The execute operation input options parameters are STACK_SIZE_ID
and PRIORITY_ID. The execute operation uses these options, when
specified, to set the operating system's process/thread stack
size and priority, for the executable image of the given input
name parameter.

The execute operation returns a unique processID for the process
that it created or a processID of minus 1 (-1) when a process is
not created.

The execute operation raises the InvalidState exception when the
Device's adminState is not UNLOCKED or operationalState is
DISABLED.

The execute operation raises the InvalidFunction exception when
the function indicated by the input name parameter does not exist
for the Device.

The execute operation raises the CF InvalidFileName exception
when the file name indicated by the input name parameter does not
exist for the Device.
```

```
      The execute operation raises the InvalidParameters exception when
      the input parameters parameter item ID and value are not string
      types.

      The execute operation raises the InvalidOptions exception when
      the input options parameter does not comply with STACK_SIZE_ID
      and  PRIORITY_ID options.
      @roseuid 3A5DAEC1016F */
      ProcessID_Type execute (
            in string name,
            in Properties options,
            in Properties parameters
            )
            raises (InvalidState, InvalidFunction, InvalidParameters,
                    InvalidOptions, InvalidFileName);

};

/* The DeviceManager interface is used to manage a set of logical
Devices and services.  The interface for a DeviceManager is based upon
its attributes, which are:

1. Device Configuration Profile - a mapping of physical device
locations to meaningful labels (e.g., audio1, serial1, etc.), along
with the Devices and services to be deployed.

2. File System - the FileSystem associated with this DeviceManager.

3. Device Manager Identifier  - the instance-unique identifier for this
DeviceManager.

4. Device Manager Label - the meaningful name given to this
DeviceManager.

5. Registered Devices - a list of Devices that have registered with
this DeviceManager.

6. Registered Services - a list of Services that have registered with
this DeviceManager.


The DeviceManager upon start up registers itself with a DomainManager.
This requirement allows the system to be developed where at a minimum
only the DomainManager's component reference needs to be known.  A
DeviceManager uses the DeviceManager's deviceConfigurationProfile
attribute for determining:

1. How to obtain the DomainManager component reference, whether Naming
Service is being used or a DomainManager stringified IOR is being used,

2. Services to be deployed for this DeviceManager (for example,
log(s)),

3. Devices to be created for this DeviceManager (when the DCD
deployondevice element is not specified then the DCD
```

componentinstantiation element is deployed on the same hardware device as the DeviceManager),

4. Devices to be deployed on (executing on) another Device,

5. Devices to be aggregated to another Device,

6. Mount point names for FileSystems,

7. The DCD's id attribute for the DeviceManager's identifier attribute value, and

8. The DCD's name attribute for the DeviceManager's label attribute value.

The DeviceManager creates FileSystem components implementing the FileSystem interface for each OS file system. If multiple FileSystems are to be created, the DeviceManager mounts created FileSystems to a FileManager component (widened to a FileSystem through the FileSys attribute). Each mounted FileSystem name must be unique within the DeviceManager.

The DeviceManager supplies execute operation parameters (IDs and format values) for a Device consisting of:

a. DeviceManager IOR - The ID is "DEVICE-_MGR_IOR" and the value is a string that is the DeviceManager stringified IOR.

b. Profile Name - The ID is "PROFILE_NAME" and the value is a CORBA string that is the full mounted file system file path name.

c. Device Identifier - The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD componentinstantiation id attribute.

d. Device Label - The ID is "DEVICE_LABEL" and the value is a string that corresponds to the DCD componentinstantiation usage element. This parameter is only used when the DCD componentinstantiation usageelement is specified.

e. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an AggregateDevice stringified IOR. This parameter is only used when the DCD componentinstantiation element is a composite part of another componentinstantiation element.

f. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element. The DeviceManager shall pass the componentinstantiation element "execparam" properties that have values as parameters.

The DeviceManager passes "execparam" parameters' IDs and values as string values.The DeviceManager uses the componentinstantiation element's SPD implementation code's stacksize and priority elements, when specified, for the execute options parameters.

The DeviceManager initializes and configures logical Devices that are started by the DeviceManager after they have registered with the

```
DeviceManager.

The DeviceManager configures a DCD's componentinstantiation element
provided the componentinstantiation element has "configure" readwrite
or writeonly properties with values.

If a Service is deployed by the DeviceManager, the DeviceManager
supplies execute operation parameters (IDs and format values)
consisting of:

a. DeviceManager IOR - The ID is "DEVICE_MGR_IOR" and the value is a
string that is the DeviceManager stringified IOR.

b. Service Name - The ID is "SERVICE_NAME" and the value is a string
that corresponds to the DCD componentinstantiation usagename element.
*/

interface DeviceManager : PropertySet, PortSupplier {
      /* This structure provides the object reference and name of
      services that have registered with the DeviceManager. */

      struct ServiceType {
            Object serviceObject;
            string serviceName;
      };

      /* This type provides an unbounded sequence of ServiceType
      structures for services that have registered with the
      DeviceManager. */

      typedef sequence <ServiceType> ServiceSequence;

      /* The readonly deviceConfigurationProfile attribute contains the
      DeviceManager's profile. The readonly deviceConfigurationProfile
      attribute contains either a profile element with a file reference
      to the DeviceManager's device configuration (DCD) profile or the
      XML for the DeviceManager's device configuration (DCD) profile.
      Files referenced within the profile are obtained from a
      FileSystem. */

      readonly attribute string deviceConfigurationProfile;
      /* The readonly fileSys attribute contains the FileSystem
      associated with this DeviceManager or a nil CORBA object
      reference if no FileSystem is associated with this DeviceManager.
      */

      readonly attribute FileSystem fileSys;
      /* The readonly identifier attribute contains the instance-unique
      identifier for a DeviceManager. */

      readonly attribute string identifier;
      /* The readonly label attribute contains the DeviceManager's
      label.  The label attribute is the meaningful name given to a
      DeviceManager. */

      readonly attribute string label;
```

```
/* The readonly registeredDevices attribute contains a list of
Devices that have registered with this DeviceManager or a
sequence length of zero if no Devices have registered with the
DeviceManager. */

readonly attribute DeviceSequence registeredDevices;
/* The readonly registeredServices attribute shall contain a list
of Services that have registered with this DeviceManager or a
sequence length of zero if no Services have registered with the
DeviceManager. */

readonly attribute ServiceSequence registeredServices;

/* This operation provides the mechanism to register a Device
with a DeviceManager.

The registerDevice operation adds the input registeringDevice to
the DeviceManager's registeredDevices attribute when the input
registeringDevice does not already exist in the registeredDevices
attribute. The registeringDevice is ignored when duplicated.

The registerDevice operation registers the registeringDevice with
the DomainManager when the DeviceManager has already registered
to the DomainManager and the registeringDevice has been
successfully added to the DeviceManager's registeredDevices
attribute.

The registerDevice operation writes a log record with the log
level set to FAILURE_ALARM, upon unsuccessful registration of a
Device to the DeviceManager's registeredDevices.


This operation does not return any value.

The registerDevice operation raises the CF InvalidObjectReference
when the input registeredDevice is a nil CORBA object reference.
@roseuid 3ACFA4F90122 */
void registerDevice (
      in Device registeringDevice
      )
      raises (InvalidObjectReference);

/* This operation unregisters a Device from a DeviceManager.

The unregisterDevice operation removes the input registeredDevice
from the DeviceManager's registeredDevices attribute.  The
unregisterDevice operation unregisters the input registeredDevice
from the DomainManager when the input registeredDevice is
registered with the DeviceManager and the DeviceManager is not
shutting down.

The unregisterDevice operation writes a log record with the log
level set to FAILURE_ALARM, when it cannot successfully remove a
registeredDevice from the DeviceManager's registeredDevices.
```

```
This operation does not return any value.

The unregisterDevice operationraises the CF
InvalidObjectReference when the input registeredDevice is a nil
CORBA object reference or does not exist in the DeviceManager's
registeredDevices attribute.
@roseuid 3ACFA5040000 */
void unregisterDevice (
      in Device registeredDevice
      )
      raises (InvalidObjectReference);

/* This operation provides the mechanism to terminate a
DeviceManager.

The shutdown operation unregisters the DeviceManager from the
DomainManager.

The shutdown operation performs releaseObject on all of the
DeviceManager's registered Devices (DeviceManager's
registeredDevices attribute).

The shutdown operation causes the DeviceManager to be unavailable
(i.e. released from the CORBA environment and its process
terminated on the OS), when all of the DeviceManager's registered
Devices are unregistered from the DeviceManager.

This operation does not return any value.

This operation does not raise any exceptions.
@roseuid 3ACFA50E0302 */
void shutdown ();

/* This operation provides the mechanism to register a Service
with a DeviceManager.

The registerService operation adds the input registeringService
to the DeviceManager's registeredServices attribute when the
input registeringService does not already exist in the
registeredServices attribute. The registeringService is ignored
when duplicated.

The registerService operation registers the registeringService
with the DomainManager when the DeviceManager has already
registered to the DomainManager and the registeringService has
been successfully added to the DeviceManager's registeredServices
attribute.

The registerService operation writes a log record with the log
level set to FAILURE_ALARM, upon unsuccessful registration of a
Service to the DeviceManager's registeredServices.

This operation does not return any value.

The registerService operation raises the CF
InvalidObjectReference exception when the input registeredService
```

```
            is a nil CORBA object reference.
            @roseuid 3B338F910156 */
            void registerService (
                  in Object registeringService,
                  in string name
                  )
                  raises (InvalidObjectReference);

            /* This operation unregisters a Service from a DeviceManager.

            The unregisterService operation removes the input
            registeredService from the DeviceManager's registeredServices
            attribute.  The unregisterService operation unregisters the input
            registeredService from the DomainManager when the input
            registeredService is registered with the DeviceManager and the
            DeviceManager is not in the shutting down state.

            The unregisterService operation writes a log record with the log
            level set to FAILURE_ALARM, when it cannot successfully remove a
            registeredService from the DeviceManager's registeredServices.

            This operation does not return any value.

            The unregisterService operation raises the CF
            InvalidObjectReference when the input registeredService is a nil
            CORBA object reference or does not exist in the DeviceManager's
            registeredServices attribute.
            @roseuid 3B338F950007 */
            void unregisterService (
                  in Object registeredService,
                  in string name
                  )
                  raises (InvalidObjectReference);

            /* This operation returns the SPD implementation ID that the
            DeviceManager interface used to create a component.

            The getComponentImplementationId operation returns the SPD
            implementation ID attribute that matches the SPD implementation
            which was used to create the component as identified by the inpu
            componentInstantiation ID parameter. The
            getComponentImplementationId operation returns an empty string
            when the componentInstantiation is invalid.

            This operation does not raise any exceptions.
            @roseuid 3B45B6E301FB */
            string getComponentImplementationId (
                  in string componentInstantiationId
                  );

      };

};

#endif
```

C.2 PortTypes MODULE.

This CORBA Module contains a set of unbundled CORBA sequence types based on CORBA types not in the CF CORBA Module. The Basic Sequence Types IDL was generated from the Rational Rose model, version 98i.

```
//## Module: PortTypes
//## Subsystem:
Core_CSCI_IDL_Implementation_Components::CF_IDL_Implementation_Component
//## Source file:
/software/components/RadioCORBA/mmits_sdr/sdr_code.ss/glbick_jtrs.wrk/PortTypes
.idl
//##begin module.cm preserve=no
//    %X% %Q% %Z% %W%
//##end module.cm

//##begin module.cp preserve=no
//##end module.cp


#ifndef PortTypes_idl
#define PortTypes_idl

//##begin module.additionalIncludes preserve=no
//##end module.additionalIncludes

//##begin module.includes preserve=yes
//##end module.includes


// ==================================================================

module PortTypes
{
  //##begin module.declarations preserve=no
  //##end module.declarations

  //##begin module.additionalDeclarations preserve=yes
  //##end module.additionalDeclarations


  //## WstringSequence Documentation:
  //      This type is a CORBA unbounded sequence of Wstrings.
  //## Category: Port_Types_IDL_Components

  typedef sequence<wstring> WstringSequence;


  //## BooleanSequence Documentation:
  //      This type is a CORBA unbounded sequence of booleans.
  //## Category: Port_Types_IDL_Components

  typedef sequence<boolean> BooleanSequence;
```

```
//## CharSequence Documentation:
//      This type is a CORBA unbounded sequence of
//      characters.
//## Category: Port_Types_IDL_Components

typedef sequence<char> CharSequence;


//## DoubleSequence Documentation:
//      This type is a CORBA unbounded sequence of doubles.
//## Category: Port_Types_IDL_Components

typedef sequence<double> DoubleSequence;


//## LongDoubleSequence Documentation:
//      This type is a CORBA unbounded sequence of long
//      Doubles.
//## Category: Port_Types_IDL_Components

typedef sequence<long double> LongDoubleSequence;


//## LongLongSequence Documentation:
//      This type is a CORBA unbounded sequence of
//      longlongs.
//## Category: Port_Types_IDL_Components

typedef sequence<long long> LongLongSequence;


//## LongSequence Documentation:
//      This type is a CORBA unbounded sequence of longs.
//## Category: Port_Types_IDL_Components

typedef sequence<long> LongSequence;


//## ShortSequence Documentation:
//      This type is a CORBA unbounded sequence of shorts.
//## Category: Port_Types_IDL_Components

typedef sequence<short> ShortSequence;


//## UlongLongSequence Documentation:
//      This type is a CORBA unbounded sequence of unsigned
//      lon longs.
//## Category: Port_Types_IDL_Components

typedef sequence<unsigned long long> UlongLongSequence;


//## UlongSequence Documentation:
//      This type is a CORBA unbounded sequence of unsigned
```

```
  //      longs.
  //## Category: Port_Types_IDL_Components

  typedef sequence<unsigned long> UlongSequence;


  //## UshortSequence Documentation:
  //      This type is a CORBA unbounded sequence of unsigned
  //      shorts.
  //## Category: Port_Types_IDL_Components

  typedef sequence<unsigned short> UshortSequence;


  //## WcharSequence Documentation:
  //      This type is a CORBA unbounded sequence of
  //      wcharacters.
  //## Category: Port_Types_IDL_Components

  typedef sequence<wchar> WcharSequence;


  //## FloatSequence Documentation:
  //      This type is a CORBA unbounded sequence of floats.
  //## Category: Port_Types_IDL_Components

  typedef sequence<float> FloatSequence;


};

#endif
```

C.3 PushPorts MODULE.

This CORBA Module contains the PushPorts interfaces where each interface is based upon one CORBA basic type. The PushPorts CORBA module contains a set of data interfaces that extend the CF::*Port* interface. Each interface is a push consumer type where a producer uses that interface to push data to a consumer. Each interface contains one "one-way" operation that is based upon a standard CORBA type such as octet, short, and long. The operation parameters are a sequence of a basic CORBA type parameter along with a control parameter. The types for these parameters are defined in PortTypes CORBA module. The requirements for these interfaces are implementation-dependent and the control information is port interface dependent. The PushPorts CORBA module can be implemented either as a CORBA TIE or non-TIE approach. The benefit of the TIE approach is it allows a push consumer servant to implement more than one of these interfaces, which is allowed by the Software Profile. This standard set of interfaces can be used by application developers for defining their CF::*Port* interfaces. The module diagram for PushPorts is shown in Figure C-3.

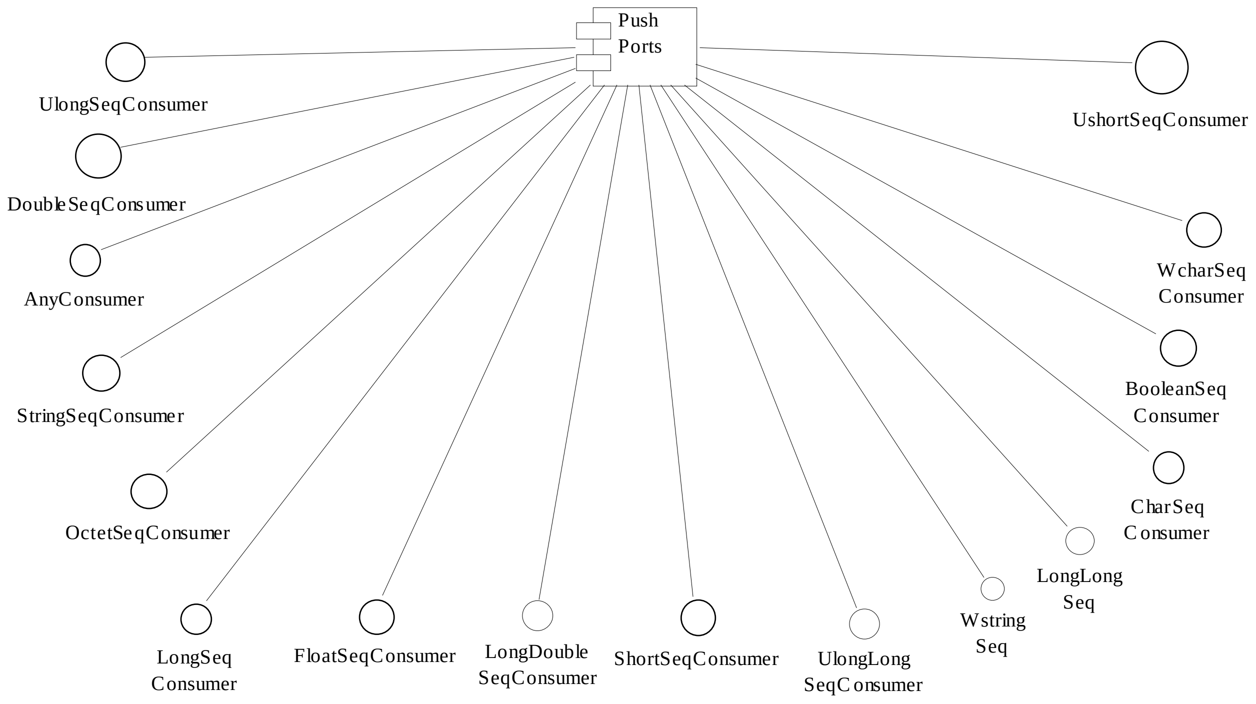

Figure C-3. PushPorts

The following is the PushPorts IDL generated from the Rational Rose model, version 98i.

```
//## Module: PushPorts
```

```
//## Subsystem:
Core_CSCI_IDL_Implementation_Components::CF_IDL_Implementation_Component
//## Source file:
/software/components/RadioCORBA/mmits_sdr/sdr_code.ss/glbick_jtrs.wrk/PushPorts
.idl
//## Documentation::
//      This CORBA Module contains the Push Port interfaces
//      where each interface is based upon one CORBA basic
//      type.
//##begin module.cm preserve=no
//    %X% %Q% %Z% %W%
//##end module.cm

//##begin module.cp preserve=no
//##end module.cp


#ifndef PushPorts_idl
#define PushPorts_idl

//##begin module.additionalIncludes preserve=no
//##end module.additionalIncludes

//##begin module.includes preserve=yes
//##end module.includes

#include "CF.idl"
#include "PortTypes.idl"

// ==================================================================

module PushPorts
{
  //##begin module.declarations preserve=no
  //##end module.declarations

  //##begin module.additionalDeclarations preserve=yes
  //##end module.additionalDeclarations


  //## OctetSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process an octet sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface OctetSeqConsumer {
    //##begin OctetSeqConsumer.initialDeclarations preserve=yes
    //##end OctetSeqConsumer.initialDeclarations

    // Attributes


    // Relationships
    // Associations


    // Operations

    //## Operation: processOctetMsg
    //## Documentation:
    //      This operation is used to push a sequence of Octets
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushConsumer) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processOctetMsg(in CF::OctetSequence msg, in CF::Properties
options);



    //##begin OctetSeqConsumer.additionalDeclarations preserve=yes
    //##end OctetSeqConsumer.additionalDeclarations

  };

  //## WcharSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a wide character sequence pushed to
  //      them by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface WcharSeqConsumer {
    //##begin WcharSeqConsumer.initialDeclarations preserve=yes
    //##end WcharSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processWcharMsg
    //## Documentation:
    //      This operation is used to push a sequence of Wchars
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processWcharMsg(in PortTypes::WcharSequence msg, in
CF::Properties options);
```

```
    //##begin WcharSeqConsumer.additionalDeclarations preserve=yes
    //##end WcharSeqConsumer.additionalDeclarations

  };

  //## LongSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a long sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface LongSeqConsumer {
    //##begin LongSeqConsumer.initialDeclarations preserve=yes
    //##end LongSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processLongMsg
    //## Documentation:
    //      This operation is used to push a sequence of Longs
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processLongMsg(in PortTypes::LongSequence msg, in
CF::Properties options);



    //##begin LongSeqConsumer.additionalDeclarations preserve=yes
    //##end LongSeqConsumer.additionalDeclarations

  };

  //## ShortSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a short sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface ShortSeqConsumer {
    //##begin ShortSeqConsumer.initialDeclarations preserve=yes
    //##end ShortSeqConsumer.initialDeclarations

    // Attributes
    // Relationships


    // Associations


    // Operations

    //## Operation: processShortMsg
    //## Documentation:
    //      This operation is used to push a sequence of Shorts
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processShortMsg(in PortTypes::ShortSequence msg, in
CF::Properties options);



    //##begin ShortSeqConsumer.additionalDeclarations preserve=yes
    //##end ShortSeqConsumer.additionalDeclarations

  };

  //## LongLongSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a CORBA long long sequence pushed to
  //      them by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface LongLongSeqConsumer {
    //##begin LongLongSeqConsumer.initialDeclarations preserve=yes
    //##end LongLongSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processLongLongMsg
    //## Documentation:
    //      This operation is used to push a sequence of Long
    //      Longs information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PushSource) object. The message
    //      being pushed has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    oneway void processLongLongMsg(in PortTypes::LongLongSequence msg, in
CF::Properties options);



    //##begin LongLongSeqConsumer.additionalDeclarations preserve=yes
    //##end LongLongSeqConsumer.additionalDeclarations

  };

  //## UlongSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process an unsigned long sequence pushed to
  //      them by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface UlongSeqConsumer {
    //##begin UlongSeqConsumer.initialDeclarations preserve=yes
    //##end UlongSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processUlongMsg
    //## Documentation:
    //      This operation is used to push a sequence of
    //      Unsigned Longs information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PushSource) object. The
    //      message being pushed has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    oneway void processUlongMsg(in PortTypes::UlongSequence msg, in
CF::Properties options);



    //##begin UlongSeqConsumer.additionalDeclarations preserve=yes
    //##end UlongSeqConsumer.additionalDeclarations

  };

  //## UlongLongSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process an unsigned long long sequence pushed
  //      to them by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface UlongLongSeqConsumer {
    //##begin UlongLongSeqConsumer.initialDeclarations preserve=yes
    //##end UlongLongSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processULongLongMsg
    //## Documentation:
    //      This operation is used to push a sequence of
    //      Unsigned Long Longs information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PushSource) object. The
    //      message being pushed has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    oneway void processULongLongMsg(in PortTypes::UlongLongSequence msg, in
CF::Properties options);



    //##begin UlongLongSeqConsumer.additionalDeclarations preserve=yes
    //##end UlongLongSeqConsumer.additionalDeclarations

  };

  //## FloatSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a float sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface FloatSeqConsumer {
    //##begin FloatSeqConsumer.initialDeclarations preserve=yes
    //##end FloatSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations
    //## Operation: processFloatMsg
    //## Documentation:
    //      This operation is used to push a sequence of floats
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processFloatMsg(in PortTypes::FloatSequence msg, in
CF::Properties options);



    //##begin FloatSeqConsumer.additionalDeclarations preserve=yes
    //##end FloatSeqConsumer.additionalDeclarations

  };

  //## DoubleSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a double sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface DoubleSeqConsumer {
    //##begin DoubleSeqConsumer.initialDeclarations preserve=yes
    //##end DoubleSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processDoubleMsg
    //## Documentation:
    //      This operation is used to push a sequence of
    //      Doubles information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PushSource) object. The message
    //      being pushed has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    oneway void processDoubleMsg(in PortTypes::DoubleSequence msg, in
CF::Properties options);



    //##begin DoubleSeqConsumer.additionalDeclarations preserve=yes
    //##end DoubleSeqConsumer.additionalDeclarations
  };

  //## LongDoubleSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a long double sequence pushed to them
  //      by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface LongDoubleSeqConsumer {
    //##begin LongDoubleSeqConsumer.initialDeclarations preserve=yes
    //##end LongDoubleSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processLongDoubleMsg
    //## Documentation:
    //      This operation is used to push a sequence of Long
    //      Doubles information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PushSource) object. The message
    //      being pushed has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    oneway void processLongDoubleMsg(in PortTypes::LongDoubleSequence msg, in
CF::Properties options);



    //##begin LongDoubleSeqConsumer.additionalDeclarations preserve=yes
    //##end LongDoubleSeqConsumer.additionalDeclarations

  };

  //## BooleanSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a boolean sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface BooleanSeqConsumer {
    //##begin BooleanSeqConsumer.initialDeclarations preserve=yes
    //##end BooleanSeqConsumer.initialDeclarations

    // Attributes
    // Relationships


    // Associations


    // Operations

    //## Operation: processBooleanMsg
    //## Documentation:
    //      This operation is used to push a sequence of
    //      Booleans information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PushSource) object. The message
    //      being pushed has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    oneway void processBooleanMsg(in PortTypes::BooleanSequence msg, in
CF::Properties options);



    //##begin BooleanSeqConsumer.additionalDeclarations preserve=yes
    //##end BooleanSeqConsumer.additionalDeclarations

  };

  //## CharSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a character sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface CharSeqConsumer {
    //##begin CharSeqConsumer.initialDeclarations preserve=yes
    //##end CharSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processCharMsg
    //## Documentation:
    //      This operation is used to push a sequence of Chars
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processCharMsg(in PortTypes::CharSequence msg, in
CF::Properties options);



    //##begin CharSeqConsumer.additionalDeclarations preserve=yes
    //##end CharSeqConsumer.additionalDeclarations

  };

  //## UshortSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process an unsigned short sequence pushed to
  //      them by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface UshortSeqConsumer {
    //##begin UshortSeqConsumer.initialDeclarations preserve=yes
    //##end UshortSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processUshortMsg
    //## Documentation:
    //      This operation is used to push a sequence of
    //      Unsigned Shorts information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PushSource) object. The
    //      message being pushed has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    oneway void processUshortMsg(in PortTypes::UshortSequence msg, in
CF::Properties options);



    //##begin UshortSeqConsumer.additionalDeclarations preserve=yes
    //##end UshortSeqConsumer.additionalDeclarations

  };

  //## StringSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a string sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components
  interface StringSeqConsumer {
    //##begin StringSeqConsumer.initialDeclarations preserve=yes
    //##end StringSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processStringMsg
    //## Documentation:
    //      This operation is used to push a CORBA string
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processStringMsg(in CF::StringSequence msg, in CF::Properties
options);



    //##begin StringSeqConsumer.additionalDeclarations preserve=yes
    //##end StringSeqConsumer.additionalDeclarations

  };

  //## WstringSeqConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process a wide string sequence pushed to them
  //      by producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface WstringSeqConsumer {
    //##begin WstringSeqConsumer.initialDeclarations preserve=yes
    //##end WstringSeqConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processWstringMsg
    //## Documentation:
```

```
    //      This operation is used to push a CORBA Wstring
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushSource) object. The message being pushed has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void processWstringMsg(in PortTypes::WstringSequence msg, in CF::Properties
options);



    //##begin WstringSeqConsumer.additionalDeclarations preserve=yes
    //##end WstringSeqConsumer.additionalDeclarations

  };

  //## AnyConsumer Documentation:
  //      This interface is implemented by push consumers
  //      that process an any sequence pushed to them by
  //      producers.
  //## Category: Push_Port_Consumer_IDL_Design_Components

  interface AnyConsumer {
    //##begin AnyConsumer.initialDeclarations preserve=yes
    //##end AnyConsumer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: processMsg
    //## Documentation:
    //      This operation is used to push a CORBA any
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PushConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    oneway void processMsg(in CF::DataType msg, in CF::Properties options);



    //##begin AnyConsumer.additionalDeclarations preserve=yes
    //##end AnyConsumer.additionalDeclarations

  };

};
```

```
#endif
```

C.4 PullPorts MODULE.

This CORBA Module contains the PullPorts interfaces where each interface is based upon one CORBA basic type. The PullPorts CORBA module contains a set of data interfaces that extend the CF::*Port* interface. Each interface is a pull producer type where a consumer uses that interface to pull data from a producer. Each interface contains one "two-way" operation which is based upon a standard CORBA type such as octet, short, and long. The operation "out" parameters are a sequence of a basic CORBA type parameter along with a control parameter. The types for these parameters are defined in PortTypes CORBA module. The requirements for these interfaces are implementation-dependent and the control information is port interface dependent. The PullPorts CORBA module can be implemented either as a CORBA TIE or non-TIE approach. This standard set of interfaces can be used by application developers for defining their CF::*Port* interfaces. The module diagram for PullPorts is shown in Figure C-4.

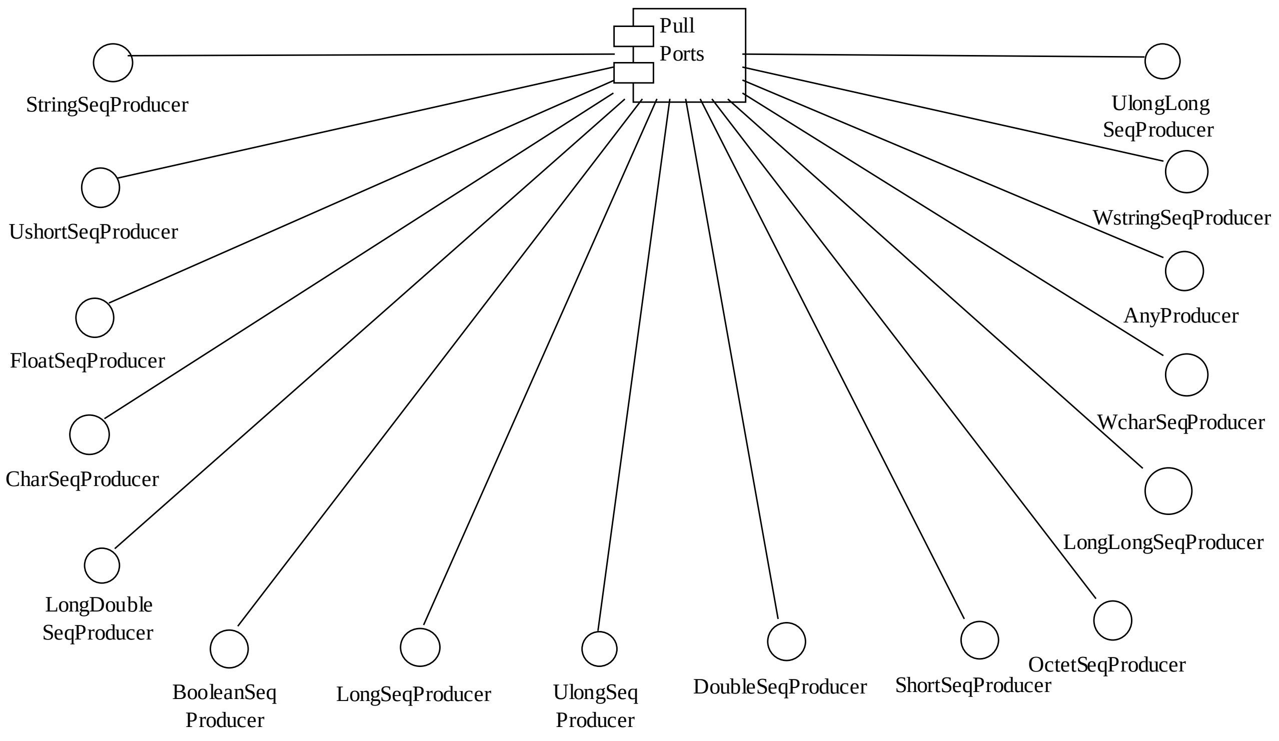

Figure C-4. PullPorts

The following is the PullPorts IDL generated from the Rational Rose model, version 98i.

```
//## Module: PullPorts
//## Subsystem:
Core_CSCI_IDL_Implementation_Components::CF_IDL_Implementation_Component
```

```
//## Source file:
/software/components/RadioCORBA/mmits_sdr/sdr_code.ss/glbick_jtrs.wrk/PullPorts
.idl
//## Documentation::
//      This CORBA Module contains the Pul Port interfaces
//      where each interface is based upon one CORBA basic
//      type.
//##begin module.cm preserve=no
//    %X% %Q% %Z% %W%
//##end module.cm

//##begin module.cp preserve=no
//##end module.cp


#ifndef PullPorts_idl
#define PullPorts_idl

//##begin module.additionalIncludes preserve=no
//##end module.additionalIncludes

//##begin module.includes preserve=yes
//##end module.includes

#include "CF.idl"
#include "PortTypes.idl"

// ================================================================

module PullPorts
{
  //##begin module.declarations preserve=no
  //##end module.declarations

  //##begin module.additionalDeclarations preserve=yes
  //##end module.additionalDeclarations


  //## OctetSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) an octet
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface OctetSeqProducer {
    //##begin OctetSeqProducer.initialDeclarations preserve=yes
    //##end OctetSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations
    // Operations

    //## Operation: getOctetMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Octets
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getOctetMsg(out CF::OctetSequence msg, out CF::Properties options);



    //##begin OctetSeqProducer.additionalDeclarations preserve=yes
    //##end OctetSeqProducer.additionalDeclarations

  };

  //## WcharSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a wide
  //      character sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface WcharSeqProducer {
    //##begin WcharSeqProducer.initialDeclarations preserve=yes
    //##end WcharSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getWcharMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Wchars
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getWcharMsg(out PortTypes::WcharSequence msg, out CF::Properties
options);



    //##begin WcharSeqProducer.additionalDeclarations preserve=yes
    //##end WcharSeqProducer.additionalDeclarations
  };

  //## LongSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a long
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface LongSeqProducer {
    //##begin LongSeqProducer.initialDeclarations preserve=yes
    //##end LongSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getLongMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Longs
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getLongMsg(out PortTypes::LongSequence msg, out CF::Properties
options);



    //##begin LongSeqProducer.additionalDeclarations preserve=yes
    //##end LongSeqProducer.additionalDeclarations

  };

  //## FloatSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a float
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface FloatSeqProducer {
    //##begin FloatSeqProducer.initialDeclarations preserve=yes
    //##end FloatSeqProducer.initialDeclarations

    // Attributes


    // Relationships
    // Associations


    // Operations

    //## Operation: getFloatMsg
    //## Documentation:
    //      This operation is used to pull a sequence of floats
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getFloatMsg(out PortTypes::FloatSequence msg, out CF::Properties
options);



    //##begin FloatSeqProducer.additionalDeclarations preserve=yes
    //##end FloatSeqProducer.additionalDeclarations

  };

  //## DoubleSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a double
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface DoubleSeqProducer {
    //##begin DoubleSeqProducer.initialDeclarations preserve=yes
    //##end DoubleSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getDoubleMsg
    //## Documentation:
    //      This operation is used to pull a sequence of
    //      Doubles information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PullConsumer) object. The message
    //      being pulled has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    void getDoubleMsg(out PortTypes::DoubleSequence msg, out CF::Properties
options);



    //##begin DoubleSeqProducer.additionalDeclarations preserve=yes
    //##end DoubleSeqProducer.additionalDeclarations

  };

  //## LongDoubleSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a long
  //      double sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface LongDoubleSeqProducer {
    //##begin LongDoubleSeqProducer.initialDeclarations preserve=yes
    //##end LongDoubleSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getLongDoubleMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Long
    //      Doubles information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PullConsumer) object. The message
    //      being pulled has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    void getLongDoubleMsg(out PortTypes::LongDoubleSequence msg, out
CF::Properties options);



    //##begin LongDoubleSeqProducer.additionalDeclarations preserve=yes
    //##end LongDoubleSeqProducer.additionalDeclarations

  };

  //## WstringSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a wide
  //      string sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components
  interface WstringSeqProducer {
    //##begin WstringSeqProducer.initialDeclarations preserve=yes
    //##end WstringSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getWstringMsg
    //## Documentation:
    //      This operation is used to pull a CORBA Wstring
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getWstringMsg(out PortTypes::WstringSequence msg, out CF::Properties
options);



    //##begin WstringSeqProducer.additionalDeclarations preserve=yes
    //##end WstringSeqProducer.additionalDeclarations

  };

  //## AnyProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) an octet
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface AnyProducer {
    //##begin AnyProducer.initialDeclarations preserve=yes
    //##end AnyProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getMsg
    //## Documentation:
```

```
    //      This operation is used to pull a CORBA any
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getMsg(out CF::DataType msg, out CF::Properties options);



    //##begin AnyProducer.additionalDeclarations preserve=yes
    //##end AnyProducer.additionalDeclarations

  };

  //## ShortSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a short
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface ShortSeqProducer {
    //##begin ShortSeqProducer.initialDeclarations preserve=yes
    //##end ShortSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getShortMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Shorts
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getShortMsg(out PortTypes::ShortSequence msg, out CF::Properties
options);



    //##begin ShortSeqProducer.additionalDeclarations preserve=yes
    //##end ShortSeqProducer.additionalDeclarations

  };

  //## BooleanSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a boolean
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface BooleanSeqProducer {
    //##begin BooleanSeqProducer.initialDeclarations preserve=yes
    //##end BooleanSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getBooleanMsg
    //## Documentation:
    //      This operation is used to pull a sequence of
    //      Booleans information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PullConsumer) object. The message
    //      being pulled has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    void getBooleanMsg(out PortTypes::BooleanSequence msg, out CF::Properties
options);



    //##begin BooleanSeqProducer.additionalDeclarations preserve=yes
    //##end BooleanSeqProducer.additionalDeclarations

  };

  //## CharSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a
  //      character sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface CharSeqProducer {
    //##begin CharSeqProducer.initialDeclarations preserve=yes
    //##end CharSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations
    // Operations

    //## Operation: getCharMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Chars
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getCharMsg(out PortTypes::CharSequence msg, out CF::Properties
options);



    //##begin CharSeqProducer.additionalDeclarations preserve=yes
    //##end CharSeqProducer.additionalDeclarations

  };

  //## LongLongSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a long
  //      long sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface LongLongSeqProducer {
    //##begin LongLongSeqProducer.initialDeclarations preserve=yes
    //##end LongLongSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getLongLongMsg
    //## Documentation:
    //      This operation is used to pull a sequence of Long
    //      Longs information to be received or transmitted
    //      through the RADIO from one object to the next
    //      "destination" (PullConsumer) object. The message
    //      being pulled has data and control information
    //      (classification, source, destination, priority,
    //      etc.).
    void getLongLongMsg(out PortTypes::LongLongSequence msg, out CF::Properties
options);
```

```
    //##begin LongLongSeqProducer.additionalDeclarations preserve=yes
    //##end LongLongSeqProducer.additionalDeclarations

  };

  //## UlongSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) an
  //      unsigned long sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface UlongSeqProducer {
    //##begin UlongSeqProducer.initialDeclarations preserve=yes
    //##end UlongSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getUlongMsg
    //## Documentation:
    //      This operation is used to pull a sequence of
    //      Unsigned Longs information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PullConsumer) object. The
    //      message being pulled has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    void getUlongMsg(out PortTypes::UlongSequence msg, out CF::Properties
options);



    //##begin UlongSeqProducer.additionalDeclarations preserve=yes
    //##end UlongSeqProducer.additionalDeclarations

  };

  //## UlongLongSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) an
  //      unsigned long long sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface UlongLongSeqProducer {
    //##begin UlongLongSeqProducer.initialDeclarations preserve=yes
    //##end UlongLongSeqProducer.initialDeclarations

    // Attributes
    // Relationships


    // Associations


    // Operations

    //## Operation: getULongLongMsg
    //## Documentation:
    //      This operation is used to pull a sequence of
    //      Unsigned Long Longs information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PullConsumer) object. The
    //      message being pulled has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    void getULongLongMsg(out PortTypes::UlongLongSequence msg, out
CF::Properties options);



    //##begin UlongLongSeqProducer.additionalDeclarations preserve=yes
    //##end UlongLongSeqProducer.additionalDeclarations

  };

  //## UshortSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) an
  //      unsigned short sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface UshortSeqProducer {
    //##begin UshortSeqProducer.initialDeclarations preserve=yes
    //##end UshortSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getUshortMsg
    //## Documentation:
    //      This operation is used to pull a sequence of
    //      Unsigned Shorts information to be received or
    //      transmitted through the RADIO from one object to
    //      the next "destination" (PullConsumer) object. The
    //      message being pulled has data and control
    //      information (classification, source, destination,
    //      priority, etc.).
    void getUshortMsg(out PortTypes::UshortSequence msg, out CF::Properties
options);



    //##begin UshortSeqProducer.additionalDeclarations preserve=yes
    //##end UshortSeqProducer.additionalDeclarations

  };

  //## StringSeqProducer Documentation:
  //      This interface is implemented by pull producers and
  //      used by a pull consumer that gets (pull) a stringt
  //      sequence from a pull producer.
  //## Category: Pull_Port_Producer_IDL_Components

  interface StringSeqProducer {
    //##begin StringSeqProducer.initialDeclarations preserve=yes
    //##end StringSeqProducer.initialDeclarations

    // Attributes


    // Relationships


    // Associations


    // Operations

    //## Operation: getStringMsg
    //## Documentation:
    //      This operation is used to pull a CORBA string
    //      information to be received or transmitted through
    //      the RADIO from one object to the next "destination"
    //      (PullConsumer) object. The message being pulled has
    //      data and control information (classification,
    //      source, destination, priority, etc.).
    void getStringMsg(out CF::StringSequence msg, out CF::Properties options);



    //##begin StringSeqProducer.additionalDeclarations preserve=yes
    //##end StringSeqProducer.additionalDeclarations

  };

};

#endif
```

C.5 LogService MODULE.

The LogService module contains the *Log* servant interface and the types necessary for a log producer to generate standard SCA log records as depicted in Figure C-5.

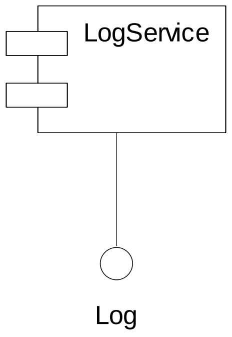

Figure C-5. LogService Module.

The following is the LogService Module IDL generated from the Rational Rose model, version 2000e.

```
#ifndef __LOGSERVICE_DEFINED
#define __LOGSERVICE_DEFINED

/* CmIdentification
  %X% %Q% %Z% %W% */

/* The LogService module contains the Log servant interface and the types
necessary for a log producer to generate standard SCA log records.
This module also defines the types necessary to control the logging output
of a log producer. Components that produce logs are required to implement
configure properties that allow the component to be configured as to what
log records it will output. */

module LogService {

      /* The LogLevelType is an enumeration type that is utilized to identify
      log levels. */

      enum LogLevelType {
            FAILURE_ALARM,
            DEGRADED_ALRAM,
            EXCEPTION_ERROR,
            FLOW_CONTROL_ERROR,
            RANGE_ERROR,
            USAGE_ERROR,
            ADMINISTRATIVE_EVENT,
            STATISTIC_REPORT,
            PROGRAMMER_DEBUG1,
            PROGRAMMER_DEBUG2,
            PROGRAMMER_DEBUG3,
            PROGRAMMER_DEBUG4,
            PROGRAMMER_DEBUG5,
            PROGRAMMER_DEBUG6,
            PROGRAMMER_DEBUG7,
            PROGRAMMER_DEBUG8,
            PROGRAMMER_DEBUG9,
            PROGRAMMER_DEBUG10,
            PROGRAMMER_DEBUG11,
            PROGRAMMER_DEBUG12,
            PROGRAMMER_DEBUG13,
            PROGRAMMER_DEBUG14,
            PROGRAMMER_DEBUG15,
            PROGRAMMER_DEBUG16
      };

      /* The LogLevelSequence type is an unbounded sequence of LogLevelTypes.
      The PRODUCER_LOG_LEVEL configure/query property is of the
      LogLevelSequence type. */

      typedef sequence <LogLevelType> LogLevelSequenceType;

      /* The ProducerLogRecordType defines the log records that sent to a
      Log. */

      struct ProducerLogRecordType {
            /* This attribute uniquely identifies the source of a log record.
            The value is unique within the Domain. The DomainManager and
            ApplicationFactory are responsible for assigning this value. */
            unsigned long producerID;
            /* This attrubute identifies the producer of a log record in
            textual format. This field is assigned by the log producer, thus
            is not unique within the Domain (e.g. - multiple instances of an
            application will assign the same name to the ProducerName field.)
            */
            string producerName;
            /* This attribute identifies the type of message being logged as
            defined by the type LogLevelType. */
            LogLevelType level;
            /* This attribute contains the informational message being
            logged. */
            string logData;
      };

      /* A Log is utilized by CF and CORBA capable application components to
      store informational messages. These informational messages are referred
      to as 'log records' in this document. The interface provides operations
      for writing log records to a Log, retrieving log records from a Log,
      control of a log,  and status of a Log. */

      interface Log {
            /* The AdministrativeStateType denotes the active logging state
            of an operational Log. When set to UNLOCKED the Log will accept
            records for storage, per its operational parameters. When set to
            LOCKED the Log will not accept new log records and records can be
            read or deleted only. */
```

```
enum AdministrativeStateType {
      LOCKED,
      UNLOCKED
};

/* The AvailabilityStatusType denotes whether or not the Log is
available for use. When true,offDuty indicates the Log is LOCKED
(administrative state) or DISABLED (operational state). When
true, logFull indicates the Log storage is full. */

struct AvailabilityStatusType {
      boolean offDuty;
      boolean logFull;
};

/* This type specifies the action that the Log should take when
itsit's internal buffers become full of data, leaving no room for
new records to be written. Wrap indicates that the Log will
overwrite the oldest log records with the newest records, as they
are written to the Log.Halt indicates that the Log will stop
logging when full. */

enum LogFullActionType {
      WRAP,
      HALT
};

/* The enumeration OperationalStateType defines the Log states of
operation. When the Log is ENABLED it is fully functional and is
available for use by log producer and log consumer clients. A Log
that is DISABLED has encountered a runtime problem and is not
available for use by log producers or log consumers. The internal
error conditions that cause the Log to set the operational state
to ENABLED or DISABLED are implementation specific. */

enum OperationalStateType {
      DISABLED,
      ENABLED
};

/* This exception indicates that the log is empty. */

exception LogEmpty {
};

exception InvalidLogFullAction {
      string Details;
};

/* This exception indicates that a provided parameter was
invalid. */

exception InvalidParam {
      string details;
};
```

```
/* This type provides the time format used when writing log
records.  The Log implementation is required to produce time-
stamps compatible with the Posix defined struct tm. */

typedef unsigned long long LogTimeType;

/* This type provides the record ID that is assigned to a log
record. */

typedef unsigned long long RecordIdType;

/* The LogRecordType defines the format of the log records as
stored in the Log.  The 'info' field is the ProducerLogRecordType
that is written by a client to the Log. */

struct LogRecordType {
      RecordIdType id;
      LogTimeType time;
      ProducerLogRecordType info;
};

/* The ProducerLogRecordSequence type defines a sequence of
ProducerLogRecordTypes. */

typedef
sequence <ProducerLogRecordType> ProducerLogRecordSequence;

/* The LogRecordSequence type defines an unbounded sequence of
log records. */

typedef sequence <LogRecordType> LogRecordSequence;

/* This operation provides the maximum number of bytes that the
Log can store.

The getMaxSizeis operation returns the integer number of bytes
that the Log is capable of storing.

This operation does not raise any exceptions.
@roseuid 3B268C6203B5 */
unsigned long long getMaxSize ();

/* This operation sets the maximum number of bytes that the Log
can store.

The setMaxSize operation sets the maximum size of the log
measured in number of bytes.

This operation does not return a value.

The setMaxSize operation raises the InvalidParam exception if the
size parameter passed in is less than the current size of the
Log.

The setMaxSizeis operation raises the InvalidParam exception if
```

```
the size parameter passed in is less than the current size of the
Log
@roseuid 3B268CAF0207 */
void setMaxSize (
      in unsigned long long size
      )
      raises (InvalidParam);

/* The getCurrentSize operation provides the current size of the
log storage in bytes.

The getCurrentSize operation returns the current size of the log
storage in bytes (i.e. if the log contains no records, get
CurrentSize will return a value of 0.).

This operation does not return any exceptions.
@roseuid 3B268D1500C4 */
unsigned long long getCurrentSize ();

/* The getNumRecords operation provides the number of records
present in the Log.

The getNumRecords operation returns the current number of log
records contained in the Log.

This operation does not raise any exceptions.
@roseuid 3B268D2B00D9 */
unsigned long long getNumRecords ();

/* The getLogFullAction operation provides the action taken when
the Log becomes full.

The getLogFullAction operation returns the Log's full action
setting.

This operation does not return any exceptions.
@roseuid 3B268D4603BD */
LogFullActionType getLogFullAction ();

/* The setLogFullAction operation is used to configure the Log to
either WRAP or HALT when the Log becomes full.

The setLogFullAction operation shall set the action taken when
the maximum size of the Log has been reached.

This operation does not return a value.

This operation does not return any exceptions.
@roseuid 3B268D6503B8 */
void setLogFullAction (
      in LogFullActionType action
      );

/* The getAvailabilityStatus operation is used to read the
availability status of the Log.
```

```
The getAvailabilityStatus operation returns the current
availability status of the Log.

This operation does not raise any exceptions.
@roseuid 3B268DD302CF */
AvailabilityStatusType getAvailabilityStatus ();

/* The getAdministrativeState is used to read the administrative
state of the Log.

The getAdministrativeState operation returns the current
administrative state of the Log.

This operation does not raise any exceptions.
@roseuid 3B268DEC0376 */
AdministrativeStateType getAdministrativeState ();

/* The setAdministrativeState operation provides write access to
the administrative state value.

The setAdministrativeState operation sets the administrative
state of the Log.

This operation does not return a value.

This operation does not raise any exceptions.
@roseuid 3B268E0503AE */
void setAdministrativeState (
      in AdministrativeStateType state
      );

/* The getOperationalState operation returns the operational
state of the Log.

This operation does not raise any exceptions.
@roseuid 3B268F0B02D8 */
OperationalStateType getOperationalState ();

/* The writeRecords operation provides the method for writing log
records to the Log. The operation is defined as oneway to
minimize client overhead while writing to the Log.

The writeRecords operation adds each log record supplied in the
records parameter to the Log.  When there is insufficient storage
to add one of the supplied log records to the Log, and the
LogFullAction is set to HALT, the writeRecords method sets the
availability status logFull state to true.  (i.e. if 3 records
are provided in the records parameter, and while trying to write
the second record to the log, the record will not fit, then the
Llog is considered to be full therefore the second and third
records will not be stored in the Log but the first record would
have been successfully stored.). The writeRecords operation
writes the current time to the time field of the LogRecord log
record in the format defined by the standard Posix type struct
tm. The writeRecords operation assigns a unique record ID to the
id field of the LogRecordlog record. Log records accepted for
```

```
storage by the writeRecords are made available for retrieval in
the order received.

This operation does not return a value.

This operation does not raise any exceptions.
@roseuid 3B32456C03B8 */
oneway void writeRecords (
      in ProducerLogRecordSequence records
      );

/* The getRecordIDFromTime operation is used to get the record ID
of the first record in the log with a time-stamp that is greater
than, or equal to, the time specified in the parameter.

The getRecordIDFromTime operation returns the record ID of the
first record in the log with a time-stamp that is greater than,
or equal to, the time specified in the fromTime parameter. The
getRecordIDFromTime operation returns zero if no record exists in
the log with a time-stamp that is greater than, or equal to,the
time specified in the fromTime parameter.

When the Log is empty, the getRecordIDFromTime operation raises
the LogEmpty exception.
@roseuid 3B33581D02C4 */
RecordIdType getRecordIdFromTime (
      in LogTimeType fromTime
      )
      raises (LogEmpty);

/* The retrieveByID operation is used to get a specified number
of records from a Log.

The retrieveByID operation returns a log record sequence that
begins with the record specified by the currentID parameter.  The
number of records in the log record sequence returned by the
retrieveByID operation is equal to the number of records
specified by the howMany parameter, or the number of records
available if the number of records specified by the howMany
parameter cannot be met.  The retrieveByID operation sets the
inout parameter currentId to the LogRecord id of next record
after the last record in the log record sequence returned.  If
the record specified by currentID does not exist, the
retrieveByID operation returns an empty list (zero) of log
records.  If the Log is empty, the retrieveByID operation returns
an empty list of log records.

This operation does not raise any exceptions.
@roseuid 3B32456D000C */
LogRecordSequence retrieveById (
      inout RecordIdType currentId,
      in unsigned long howMany
      );

/* The clearLog operation provides the method for removing all of
the log records from the Log.
```

```
            The clearLog operation deletes all records from the Log.  The
            clearLog operation sets the current size of the Log storage to
            zero.  The clearLog operation sets the current number of records
            in the Log to zero.  The clearLog operation sets the logFull
            availability status element to false.

            The clearLog operation does not return a value.

            This operation does not raise any exceptions.
            @roseuid 3B32456D003E */
            void clearLog ();

            /* The destroy operation provides a means by which an
            instantiated Log may be torn down.

            The destroy operation releases all internal memory and/or storage
            allocated by theLog.  The destroy operation tears down the
            component (i.e. released from the CORBA environment).

            The destroy operation does not return a value.

            This operation does not raise any exceptions.
            @roseuid 3B32456D007A */
            void destroy ();

      };

};

#endif
```